Con base en el ANSI C

SEGUNDA EDICION

EL LENGUAJE DE PROGRAMACION C

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

Pearson Educación ®

EDICION EN INGLES

UNIX es una marca registrada AT & T

EL LENGUAJE DE PROGRAMACION C

Traducido de la segunda edición en inglés de:

THE C PROGRAMMING LANGUAGE

Atlacomulco Núm. 500-5° Piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México

Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1524

ISBN 968-880-205-0

ISBN 0-13-110362-8

IMPRESO EN MEXICO / PRINTED IN MEXICO

CONTENIDO

Prefacio **ix**

Prefacio a la primera edición **xi**

Introducción **1**

Capítulo 1. Introducción general **5**
- 1.1 Comencemos 5
- 1.2 Variables y expresiones aritméticas 8
- 1.3 La proposición for 14
- 1.4 Constantes simbólicas 15
- 1.5 Entrada y salida de caracteres 16
- 1.6 Arreglos 23
- 1.7 Funciones 26
- 1.8 Argumentos—llamada por valor 29
- 1.9 Arreglos de caracteres 30
- 1.10 Variables externas y alcance 33

Capítulo 2. Tipos, operadores y expresiones **39**
- 2.1 Nombres de variables 39
- 2.2 Tipos y tamaños de datos 40
- 2.3 Constantes 41
- 2.4 Declaraciones 44
- 2.5 Operadores aritméticos 45
- 2.6 Operadores de relación y lógicos 46
- 2.7 Conversiones de tipo 47
- 2.8 Operadores de incremento y decremento 51
- 2.9 Operadores para manejo de bits 53
- 2.10 Operadores de asignación y expresiones 55
- 2.11 Expresiones condicionales 56
- 2.12 Precedencia y orden de evaluación 57

Capítulo 3. Control de flujo **61**
- 3.1 Proposiciones y bloques 61
- 3.2 If-else 61
- 3.3 Else-if 63
- 3.4 Switch 64
- 3.5 Ciclos—while y for 66
- 3.6 Ciclos—do-while 70

3.7 Break y continue 71
3.8 Goto y etiquetas 72

Capítulo 4. Funciones y la estructura del programa **75**
4.1 Conceptos básicos de funciones 75
4.2 Funciones que regresan valores no-enteros 79
4.3 Variables externas 82
4.4 Reglas de alcance 88
4.5 Archivos header 90
4.6 Variables estáticas 91
4.7 Variables registro 92
4.8 Estructura de bloque 93
4.9 Inicialización 94
4.10 Recursividad 95
4.11 El preprocesador de C 97

Capítulo 5. Apuntadores y arreglos **103**
5.1 Apuntadores y direcciones 103
5.2 Apuntadores y argumentos de funciones 105
5.3 Apuntadores y arreglos 108
5.4 Aritmética de direcciones 111
5.5 Apuntadores a caracteres y funciones 115
5.6 Arreglos de apuntadores; apuntadores a apuntadores 118
5.7 Arreglos multidimensionales 122
5.8 Inicialización de arreglos de apuntadores 124
5.9 Apuntadores vs. arreglos multidimensionales 125
5.10 Argumentos en la línea de comandos 126
5.11 Apuntadores a funciones 130
5.12 Declaraciones complicadas 134

Capítulo 6. Estructuras **141**
6.1 Conceptos básicos sobre estructuras 141
6.2 Estructuras y funciones 143
6.3 Arreglos de estructuras 146
6.4 Apuntadores o estructuras 151
6.5 Estructuras autorreferenciadas 153
6.6 Búsqueda en tablas 158
6.7 Typedef 161
6.8 Uniones 162
6.9 Campos de bits 164

Capítulo 7. Entrada y salida **167**
7.1 Entrada y salida estándar 167
7.2 Salida con formato—printf 169
7.3 Listas de argumentos de longitud variable 171
7.4 Entrada con formato—scanf 173
7.5 Acceso a archivos 176
7.6 Manejo de errores—stderr y exit 179
7.7 Entrada y salida de líneas 181
7.8 Otras funciones 183

Capítulo 8. La interfaz del sistema UNIX **187**
8.1 Descriptores de archivos 187
8.2 E/S de bajo nivel—read y write 188
8.3 open, creat, close, unlink 190
8.4 Acceso aleatorio—lseek 193
8.5 Ejemplo—una implementación de fopen y getc 194
8.6 Ejemplo—listado de directorios 198
8.7 Ejemplo—asignador de memoria 204

Apéndice A. Manual de referencia **211**
A1 Introducción 211
A2 Convenciones léxicas 211
A3 Notación sintáctica 215
A4 Significado de los identificadores 215
A5 Objetos y valores-l 217
A6 Conversiones 217
A7 Expresiones 220
A8 Declaraciones 232
A9 Proposiciones 245
A10 Declaraciones externas 249
A11 Alcance y ligadura 251
A12 Preprocesamiento 253
A13 Gramática 258

Apéndice B. Biblioteca estándar **265**
B1 Entrada y salida: <stdio.h> 265
B2 Pruebas de clasificación de caracteres: <ctype.h> 272
B3 Funciones para cadenas: <string.h> 273
B4 Funciones matemáticas: <math.h> 274
B5 Funciones de utilería: <stdlib.h> 275
B6 Diagnósticos: <assert.h> 278
B7 Listas de argumentos variables: <stdarg.h> 278
B8 Saltos no locales: <setjmp.h> 278
B9 Señales: <signal.h> 279
B10 Funciones de fecha y hora: <time.h> 279
B11 Límites definidos en la implantación: <limits.h> y <float.h> 281

Apéndice C. Resumen de modificaciones **283**

Indice **287**

Prefacio

El mundo de la computación ha sufrido una revolución desde la publicación, en 1978, de *El lenguaje de programación C*. Las grandes computadoras son ahora mucho más grandes, y las computadoras personales tienen capacidades que rivalizan con los *mainframes* de hace una década. También el lenguaje C ha cambiado en ese tiempo, aunque sólo en forma modesta, y se ha extendido más allá de lo que fueron sus orígenes como el lenguaje del sistema operativo UNIX.

La creciente popularidad de C, los cambios en el lenguaje a lo largo de los años, y la creación de compiladores por grupos no involucrados en su diseño, se combinaron para demostrar la necesidad de una definición del lenguaje más precisa y contemporánea que la que proporcionó la primera edición de este libro. En 1983, el *American National Standards Institute* (ANSI) estableció un comité cuyos propósitos eran producir "una definición no ambigua del lenguaje C e, independiente de la máquina", cuidando la conservación de su espíritu. El resultado es el estándar ANSI para el lenguaje C.

El estándar formaliza construcciones sugeridas pero no descritas en la primera edición, particularmente la asignación de estructura y las enumeraciones. Proporciona una nueva forma de declaración de funciones, que permite revisar comparativamente su definición y uso. Especifica una biblioteca estándar, con un conjunto extensivo de funciones para realizar la entrada y salida, la administración de memoria, la manipulación de cadenas y tareas semejantes. Precisa el comportamiento de características que no se mencionaron en la definición original, y al mismo tiempo establece explícitamente cuáles aspectos del lenguaje tienen aún dependencia de máquina.

Esta segunda edición de *El lenguaje de programación C* lo describe tal como lo definió el estándar ANSI. (En el momento de escribir está edición, el estándar se encontraba en la etapa final de revisión; se esperaba su aprobación a finales de 1988. Las diferencias entre lo que se ha descrito aquí y la forma final deberán ser mínimas.) Aunque hemos hecho anotaciones en los lugares donde el lenguaje ha evolucionado, preferimos escribir exclusivamente en la nueva forma. En general esto no hace una diferencia significativa; el cambio más visible es la nueva forma de declaración y definición de funciones. Los modernos compiladores manejan ya la mayoría de las posibilidades del estándar.

Hemos tratado de mantener la brevedad de la primera edición. El lenguaje C no es grande, y no le está bien un gran libro. Hemos mejorado la exposición de características críticas, como los apuntadores, que son parte central en la programación con C. Hemos redefinido los ejemplos originales y agregamos ejemplos nuevos en varios capítulos. Por ejemplo, se aumentó el tratamiento de declaraciones complicadas con programas que convierten declaraciones en palabras y viceversa. Como antes, todos los ejemplos se han probado directamente a partir del texto, el cual está diseñado de manera que lo pueda leer la máquina.

El apéndice A, manual de referencia, no es el estándar, sino que nuestra intención fue trasladar la esencia del estándar a un espacio más pequeño. Está hecho con el ánimo de que proporcione una fácil comprensión para los programadores, pero no como una definición del lenguaje para quienes escriben compiladores —ese papel propiamente le corresponde al estándar en sí. El apéndice B es un resumen de las posibilidades de la biblioteca estándar. También tiene el propósito de ser una referencia para programadores, no para implantadores. En el apéndice C se ofrece un resumen de los cambios de la versión original.

Como mencionamos en el prefacio a la primera edición, C "se lleva bien, en la medida en que aumenta nuestra experiencia con él". Con una década más de experiencia, aún lo sentimos así. Deseamos que este libro le ayude a aprender el lenguaje C y también cómo usarlo.

Tenemos un profundo reconocimiento hacia los amigos que nos ayudaron a producir esta segunda edición. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson y Rob Pike nos dieron valiosos comentarios sobre casi cada página del borrador de este manuscrito. Estamos agradecidos por la cuidadosa lectura de Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, y Chris Van Wyk. También recibimos útiles sugerencias de Bill Cheswick, Mark Kernighan, Andy Koening, Robin Lake, Tom London, Jim Reeds, Clovis Tondo y Peter Weinberger. Dave Prosser respondió muchas preguntas detalladas acerca del estándar ANSI. Utilizamos extensivamente el intérprete de C++ de Bjarne Stroustrup, para la prueba local de nuestros programas, y Dave Kristol nos ofreció un compilador ANSI C para las pruebas finales. Rich Drechsler nos ayudó grandemente con la composición.

Nuestro sincero agradecimiento a todos.

Brian W. Kernighan
Dennis M. Ritchie

Prefacio a la primera edición

C es un lenguaje de programación de propósito general que ofrece como ventajas economía de expresión, control de flujo y estructuras de datos modernos y un rico conjunto de operadores. Además, C no es un lenguaje de "muy alto nivel" ni "grande", y no está especializado en alguna área especial de aplicación. Pero su ausencia de restricciones y su generalidad lo hacen más conveniente y efectivo para muchas tareas que otros lenguajes supuestamente más poderosos. Originalmente, C fue diseñado para el sistema operativo UNIX y Dennis Ritchie lo implantó sobre el mismo en la DEC PDP-11. El sistema operativo, el compilador de C y esencialmente todos los programas de aplicación de UNIX (incluyendo todo el sofware utilizado para preparar este libro) están escritos en C. También existen compiladores para la producción en otras máquinas, incluyendo la IBM System/370, la Honeywell 6000 y la Interdata 8/32. El lenguaje C no está ligado a ningún hardware o sistema en particular y es fácil escribir programas que correrán sin cambios en cualquier máquina que maneje C.

La finalidad de este libro es ayudar al lector a aprender cómo programar en C. Contiene una introducción general para hacer que los nuevos usuarios se inicien lo más pronto posible, capítulos separados sobre cada característica importante y un manual de referencia. La mayoría de las exposiciones están basadas en la lectura, escritura y revisión de ejemplos, más que en el simple establecimiento de reglas. En su mayoría, los ejemplos son programas reales y completos, no fragmentos aislados. Todos los ejemplos han sido probados directamente a partir del texto, el cual está en forma legible para la máquina. Además de demostrar cómo hacer un uso efectivo del lenguaje, donde ha sido posible, tratamos de ilustrar algoritmos útiles y principios de buen estilo y diseño.

El libro no es un manual de introducción a la programación; se supone en él familiaridad con los conceptos básicos de programación, como variables, proposiciones de asignación, ciclos y funciones. No obstante, un programador novato deber ser capaz de leer y obtener los conceptos del lenguaje, aunque le ayudaría la cooperación de un colega más experimentado.

De acuerdo con nuestra experiencia, C ha demostrado ser un lenguaje agradable, expresivo y versátil para una amplia variedad de programas. Es fácil de aprender y se obtienen mejores resultados a medida que aumenta nuestra experiencia con él. Deseamos que este libro le ayude al lector a usarlo correctamente.

Las críticas y sugerencias de muchos amigos y colegas han aumentado muchísimo los conceptos de este libro y ha sido un placer escribirlo. En particular nuestro agradecimiento a Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy, Bill Roome, Bob Rosin y Larry Rosler que leyeron cuidadosamente las numerosas versiones. También agradecemos Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harrris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson y Peter Weinberger por sus valiosos comentarios a través de varias etapas; a Mike Lesk y Joe Ossanna, por su invaluable ayuda en la impresión.

Brian W. Kernighan
Dennis M. Ritchie

Introducción

C es un lenguaje de programación de propósito general que ha sido estrechamente asociado con el sistema UNIX en donde fue desarrollado puesto que tanto el sistema como los programas que corren en él están escritos en lenguaje C. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina, y aunque se le llama "lenguaje de programación de sistemas" debido a su utilidad para escribir compiladores y sistemas operativos, se utiliza con igual eficacia para escribir importantes programas en diversas disciplinas.

Muchas de las ideas importantes de C provienen del lenguaje BCPL, desarrollado por Martin Richards. La influencia de BCPL sobre C se continuó indirectamente a través del lenguaje B, el cual fue escrito por Ken Thompson en 1970 para el primer sistema UNIX de la DEC PDP-7.

BCPL y B son lenguajes "carentes de tipos". En contraste, C proporciona una variedad de tipos de datos. Los tipos fundamentales son caracteres, enteros y números de punto flotante de varios tamaños. Además, existe una jerarquía de tipos de datos derivados, creados con apuntadores, arreglos, estructuras y uniones. Las expresiones se forman a partir de operadores y operandos; cualquier expresión, incluyendo una asignación o una llamada a función, puede ser una proposición. Los apuntadores proporcionan una aritmética de direcciones independiente de la máquina.

C proporciona las construcciones fundamentales de control de flujo que se requieren en programas bien estructurados: agrupación de proposiciones, toma de decisiones (if-else), selección de un caso entre un conjunto de ellos (switch), iteración con la condición de paro en la parte superior (while, for) o en la parte inferior (do), y terminación prematura de ciclos (break).

Las funciones pueden regresar valores de tipos básicos, estructuras, uniones o apuntadores. Cualquier función puede ser llamada recursivamente. Las variables locales son normalmente "automáticas", o creadas de nuevo con cada invocación. La definición de una función no puede estar anidada, pero las variables pueden estar declaradas en una modalidad estructurada por bloques. Las funciones de un programa en C pueden existir en archivos fuente separados, que se compilan de manera separada. Las variables pueden ser internas a una función, externas pero conocidas sólo dentro de un archivo fuente, o visibles al programa completo.

Un paso de preprocesamiento realiza substitución de macros en el texto del programa, inclusión de otros archivos fuente y compilación condicional.

C es un lenguaje de relativo "bajo nivel". Esta caracterización no es peyorativa, simplemente significa que C trata con el mismo tipo de objetos que la mayoría de las computadoras, llámense caracteres, números y direcciones. Estos pueden ser combinados y cambiados de sitio con los operadores aritméticos y lógicos implantados por máquinas reales.

C no proporciona operaciones para tratar directamente con objetos compuestos, tales como cadenas de caracteres, conjuntos, listas o arreglos. No existen operaciones que manipulen un arreglo o una cadena completa, aunque las estructuras pueden copiarse como una unidad. El lenguaje no define ninguna facilidad para asignación de almacenamiento que no sea la de definición estática y la disciplina de pilas provista por las variables locales de funciones; no emplea *heap* ni recolector de basura. Finalmente, C en sí mismo no proporciona capacidades de entrada/salida; no hay proposiciones READ o WRITE, ni métodos propios de acceso a archivos. Todos esos mecanismos de alto nivel deben ser proporcionados por funciones llamadas explícitamente.

De manera semejante, C solamente ofrece un control de flujo franco, y lineal: condiciones, ciclos, agrupamientos y subprogramas, pero no multiprogramación, operaciones paralelas, sincronización ni cor-rutinas.

Aunque la ausencia de alguna de esas capacidades puede parecer como una grave deficiencia ("¿significa que se tiene que llamar a una función para comparar dos cadenas de caracteres?"), el mantener al lenguaje de un tamaño modesto tiene beneficios reales. Puesto que C es relativamente pequeño, se puede describir en un pequeño espacio y aprenderse con rapidez. Un programador puede razonablemente esperar conocer, entender y utilizar en verdad la totalidad del lenguaje.

Por muchos años, la definición de C fue el manual de referencia de la primera edición de *El lenguaje de programación C*. En 1983, el *American National Standards Institute* (ANSI) estableció un comité para proporcionar una moderna y comprensible definición de C. La definición resultante, el estándar ANSI o "ANSI C", se esperaba fuera aprobada a fines de 1988. La mayoría de las características del estándar ya se encuentran soportadas por compiladores modernos.

El estándar está basado en el manual de referencia original. El lenguaje ha cambiado relativamente poco; uno de los propósitos del estándar fue asegurar que la mayoría de los programas existentes pudiesen permanecer válidos o, al menos, que los compiladores pudieran producir mensajes de advertencia acerca del nuevo comportamiento.

Para la mayoría de los programadores, el cambio más importante es una nueva sintaxis para declarar y definir funciones. Una declaración de función ahora puede incluir una descripción de los argumentos de la función; la sintaxis de la definición cambia para coincidir. Esta información extra permite que los compiladores detecten más fácilmente los errores causados por argumentos que no coinciden; de acuerdo con nuestra experiencia, es una adición muy útil al lenguaje.

Existen otros cambios de menor escala en el lenguaje. La asignación de estructuras y enumeraciones, que ha estado ampliamente disponible, es ahora parte oficial del lenguaje. Los cálculos de punto flotante pueden ahora realizarse con precisión sencilla. Las propiedades de la aritmética, especialmente para tipos sin signo, están esclarecidas. El preprocesador es más elaborado. La mayor parte de esos cambios sólo tendrán efectos secundarios para la mayoría de los programadores.

Una segunda contribución significativa dei estándar es la definición de una biblioteca que acompañe a C. Esta especifica funciones para tener acceso al sistema operativo (por ejemplo, leer de archivos y escribir en ellos), entrada y salida con formato, asignación de memoria, manipulación de cadenas y otras actividades semejantes. Una colección de encabezadores (*headers*) estándar proporcionan un acceso uniforme a las declaraciones de funciones y tipos de datos. Los programas que utilizan esta biblioteca para interactuar con un sistema anfitrión están asegurados de un comportamiento compatible. La mayor parte de la biblioteca está estrechamente modelada con base en la "biblioteca E/S estándar" del sistema UNIX. Esta biblioteca se describió en la primera edición y ha sido también ampliamente utilizada en otros sistemas. De nuevo, la mayoría de los programadores no notarán mucho el cambio.

Debido a que los tipos de datos y estructuras de control provistas por C son manejadas directamente por la mayoría de las computadoras, la biblioteca de ejecución (*run-time*) requerida para implantar programas autocontenidos es pequeña. Las funciones de la biblioteca estándar únicamente se llaman en forma explícita, de manera que se pueden evitar cuando no se necesitan. La mayor parte puede escribirse en C, y excepto por detalles ocultos del sistema operativo, ellas mismas son portátiles.

Aunque C coincide con las capacidades de muchas computadoras, es independiente de cualquier arquitectura. Con un poco de cuidado es fácil escribir programas portátiles, esto es, programas que puedan correr sin cambios en una variedad de máquinas. El estándar explica los problemas de la transportabilidad, y prescribe un conjunto de constantes que caracterizan a la máquina en la que se ejecuta el programa.

C no es un lenguaje fuertemente tipificado, sino que, al evolucionar, su verificación de tipos ha sido reforzada. La definición original de C desaprobó, pero permitió, el intercambio de apuntadores y enteros; esto se ha eliminado y el estándar ahora requiere la adecuada declaración y la conversión explícita que ya ha sido obligada por los buenos compiladores. La nueva declaración de funciones es otro paso en esta dirección. Los compiladores advertirán de la mayoría de los errores de tipo, y no hay conversión automática de tipos de datos incompatibles. Sin embargo, C mantiene la filosofía básica de que los programadores saben lo que están haciendo; sólo requiere que establezcan sus intenciones en forma explícita.

Como cualquier otro lenguaje, C tiene sus defectos. Algunos de los operadores tienen la precedencia equivocada; algunos elementos de la sintaxis pueden ser

mejores. A pesar de todo, C ha probado ser un lenguaje extremadamente efectivo y expresivo para una amplia variedad de programas de aplicación.

El libro está organizado como sigue. El capítulo 1 es una introducción orientada a la parte central de C. El propósito es hacer que el lector se inicie tan pronto como le sea posible, puesto que creemos firmemente que la forma de aprender un nuevo lenguaje es escribir programas en él. La introducción supone un conocimiento práctico de los elementos básicos de la programación; no hay una explicación de computadoras, de compilación, ni del significado de una expresión como n = n + 1. Aunque hemos tratado de mostrar técnicas útiles de programación en donde fue posible, la intención del libro no es la de ser un texto de consulta sobre estructuras de datos y algoritmos; cuando nos vimos forzados a hacer una elección, nos hemos concentrado en el lenguaje.

En los capítulos del 2 al 6 se discuten varios aspectos de C en mayor detalle y más formalmente de lo que se hace en el capítulo 1, aunque el énfasis está aún en los ejemplos de programas completos, más que en fragmentos aislados. El capítulo 2 trata de los tipos básicos de datos, operaciones y expresiones. El capítulo 3 trata sobre control de flujo: **if-else**, **switch**, **while**, **for**, etc. En el capítulo 4 se cubren funciones y la estructura de un programa —variables externas, reglas de alcance, archivos fuente múltiples y otros aspectos— y también abarca al preprocesador. El capítulo 5 discute sobre apuntadores y aritmética de direcciones. El capítulo 6 cubre estructuras y uniones.

El capítulo 7 describe la biblioteca estándar, la cual proporciona una interfaz común con el sistema operativo. Esta biblioteca está definida por el estándar ANSI y se intenta que se tenga en todas las máquinas que manejan C; así, los programas que la usen para entrada, salida y otros accesos al sistema operativo se puedan transportar de un sistema a otro sin cambios.

El capítulo 8 describe una interfaz entre los programas en C y el sistema operativo UNIX, concentrándose en entrada/salida, el sistema de archivos y la asignación de memoria. Aunque algo de este capítulo es específico de sistemas UNIX, los programadores que usen otros sistemas de todas maneras encontrarán aquí material de utilidad, incluyendo alguna comprensión acerca de cómo está implantada una versión de la biblioteca estándar, así como sugerencias para obtener un código portátil.

El apéndice A contiene un manual de consulta del lenguaje. El informe oficial de la sintaxis y la semántica de C es en sí el estándar ANSI. Ese documento, sin embargo, está principalmente pensado para quienes escriben compiladores. El manual de consulta de este libro transmite la definición del lenguaje en una forma más concisa y sin el mismo estilo legalista. El apéndice B es un resumen de la biblioteca estándar, de nuevo más para usuarios que para implantadores. El apéndice C es un breve resumen de los cambios del lenguaje original. Aunque, en caso de duda, el estándar y el compilador en uso quedan como las autoridades finales sobre el lenguaje.

CAPITULO 1: Introducción general

Comencemos con una introducción rápida a C. Nuestro objetivo es mostrar los elementos esenciales del lenguaje en programas reales, pero sin perdernos en detalles, reglas o excepciones. Por el momento, no intentamos ser completos ni precisos (exceptuando en los ejemplos, que sí lo son). Deseamos llevarlo tan rápido como sea posible al punto en donde pueda escribir programas útiles, y para hacerlo tenemos que concentrarnos en las bases: variables y constantes, aritmética, control de flujo, funciones y los rudimentos de entrada y salida. Hemos dejado intencionalmente fuera de este capítulo las características de C que son importantes para escribir programas más grandes. Esas características incluyen apuntadores, estructuras, la mayor parte del rico conjunto de operadores de C, varias proposiciones para control de flujo y la biblioteca estándar.

Este enfoque tiene sus inconvenientes. Lo más notorio es que aquí no se encuentra la descripción completa de ninguna característica particular del lenguaje, y la introducción, por su brevedad, puede también ser confusa. Y debido a que los ejemplos no utilizan la potencia completa de C, no son tan concisos y elegantes como podrían serlo. Hemos tratado de aminorar esos efectos, pero tenga cuidado. Otro inconveniente es que los capítulos posteriores necesariamente repetirán algo de lo expuesto en éste. Esperamos que la repetición, más que molestar, ayude.

En cualquier caso, los programadores con experiencia deben ser capaces de extrapolar del material que se encuentra en este capítulo a sus propias necesidades de programación. Los principiantes deben complementarlo escribiendo pequeños programas semejantes a los aquí expuestos. Ambos grupos pueden utilizar este capítulo como un marco de referencia sobre el cual asociar las descripciones más detalladas que comienzan en el capítulo 2.

1.1 Comencemos

La única forma de aprender un nuevo lenguaje de programación es escribiendo programas con él. El primer programa por escribir es el mismo para todos los lenguajes:

Imprima las palabras
hola, mundo

Este es el gran obstáculo; para librarlo debe tener la habilidad de crear el texto del programa de alguna manera, compilarlo con éxito, cargarlo, ejecutarlo y descubrir a dónde fue la salida. Con el dominio de estos detalles mecánicos, todo lo demás es relativamente fácil.

En C, el programa para escribir "**hola, mundo**" es

```
#include <stdio.h>

main( )
{
      printf("hola, mundo\n");
}
```

La forma de ejecutar este programa depende del sistema que se esté utilizando. Como un ejemplo específico, en el sistema operativo UNIX se debe crear el programa en un archivo cuyo nombre termine con ".c", como **hola.c**, y después compilarlo con la orden

```
cc hola.c
```

Si no se ha cometido algún error, como la omisión de un carácter o escribir algo en forma incorrecta, la compilación se hará sin emitir mensaje alguno, y creará un archivo ejecutable llamado **a.out**. Si se ejecuta **a.out** escribiendo la orden

```
a.out
```

se escribirá

```
hola, mundo
```

En otros sistemas, las reglas serán diferentes, consúltelo con un experto.

Ahora algunas explicaciones acerca del programa en sí. Un programa en C, cualquiera que sea su tamaño, consta de *funciones* y *variables*. Una función contiene *proposiciones* que especifican las operaciones de cálculo que se van a realizar, y las variables almacenan los valores utilizados durante los cálculos. Las funciones de C son semejantes a las subrutinas y funciones de Fortran o a los procedimientos y funciones de Pascal. Nuestro ejemplo es una función llamada main. Normalmente se tiene la libertad de dar cualquier nombre que se desee, pero "**main**" es especial —el programa comienza a ejecutarse al principio de main. Esto significa que todo programa debe tener un main en algún sitio.

Por lo común main llamará a otras funciones que ayuden a realizar su trabajo, algunas que usted ya escribió, y otras de bibliotecas escritas previamente. La primera línea del programa.

```
#include <stdio.h>
```

indica al compilador que debe incluir información acerca de la biblioteca estándar de entrada/salida; esta línea aparece al principio de muchos archivos fuente de C. La biblioteca estándar está descrita en el capítulo 7 y en el apéndice B.

Un método para comunicar datos entre las funciones es que la función que llama proporciona una lista de valores, llamados *argumentos*, a la función que está invocando. Los paréntesis que están después del nombre de la función encierran a la lista de argumentos. En este ejemplo, **main** está definido para ser una función que no espera argumentos, lo cual está indicado por la lista vacía ( ).

```
# include <stdio.h>          incluye información acerca de la biblioteca estándar

main( )                      define una función llamada main
                             que no recibe valores de argumentos
{                            las proposiciones de main están encerradas entre llaves
     printf("hola, mundo\n");    main llama a la función de biblioteca printf
}                            para escribir esta secuencia de caracteres;
                             \n representa el carácter nueva línea
```

El primer programa en C

Las proposiciones de una función están encerradas entre llaves { }. La función **main** sólo contiene una proposición,

```
printf ("hola, mundo\n");
```

Una función se invoca al nombrarla, seguida de una lista de argumentos entre paréntesis; de esta manera se está llamando a la función printf con el argumento "**hola, mundo\n**". printf es una función de biblioteca que escribe la salida, en este caso la cadena de caracteres que se encuentra entre comillas.

A una secuencia de caracteres entre comillas, como "hola, mundo\n", se le llama *cadena de caracteres* o *constante de cadena*. Por el momento, nuestro único uso de cadenas de caracteres será como argumentos para printf y otras funciones.

La secuencia \n en la cadena representa el carácter *nueva línea* en la notación de C, y hace avanzar la impresión al margen izquierdo de la siguiente línea. Si se omite el **\n** (un experimento que vale la pena), encontrará que no hay avance de línea después de la impresión. Se debe utilizar \n para incluir un carácter nueva línea en el argumento de **printf**; si se intenta algo como

```
printf ("hola, mundo
");
```

el compilador de C producirá un mensaje de error.

printf nunca proporciona una nueva línea automáticamente, de manera que se pueden utilizar varias llamadas para construir una línea de salida en etapas. Nuestro primer programa también pudo haber sido escrito de la siguiente manera.

```
#include <stdio.h>

main( )
{
        printf("hola,");
        printf("mundo");
        printf("\n");
}
```

produciéndose una salida idéntica.

Nótese que \n representa un solo carácter. Una *secuencia de escape* como \n proporciona un mecanismo general y extensible para representar caracteres invisibles o difíciles de escribir. Entre otros que C proporciona están \t para tabulación, \b para retroceso, \" para comillas, y \\ para la diagonal invertida. Hay una lista completa en la sección 2.3.

Ejercicio 1-1. Ejecute el programa "hola, mundo" en su sistema. Experimente con la omisión de partes del programa, para ver qué mensajes de error se obtienen. □

Ejercicio 1-2. Experimente el descubrir qué pasa cuando la cadena del argumento de printf contiene \c, en donde c es algún carácter no puesto en lista anteriormente. □

1.2 Variables y expresiones aritméticas

El siguiente programa utiliza la fórmula $°C = (5/9)(°F-32)$ para imprimir la siguiente tabla de temperaturas Fahrenheit y sus equivalentes centígrados o Celsius:

```
0   -17
20  -6
40  4
60  15
80  26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
```



```
240 115
260 126
280 137
300 148
```

En sí el programa aún consiste de la definición de una única función llamada main. Es más largo que el que imprime "hola, mundo", pero no es complicado. Introduce varias ideas nuevas, incluyendo comentarios, declaraciones, variables, expresiones aritméticas, ciclos y salida con formato.

```
#include <stdio.h>
/* imprime la tabla Fahrenheit-Celsius
    para fahr = 0, 20, ..., 300 */
main ( )
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;      /* límite inferior de la tabla de temperaturas */
    upper = 300;    /* límite superior */
    step = 20;      /* tamaño del incremento */

    fahr = lower;
    while (fahr < = upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Las dos líneas

```
/* imprime la tabla Fahrenheit-Celsius
    para fahr = 0, 20, ..., 300 */
```

son un *comentario*, que en este caso explica brevemente lo que hace el programa. Cualesquier caracteres entre /* y */ son ignorados por el compilador, y pueden ser utilizados libremente para hacer a un programa más fácil de entender. Los comentarios pueden aparecer en cualquier lugar donde puede colocarse un espacio en blanco, un tabulador o nueva línea.

En C, se deben declarar todas las variables antes de su uso, generalmente al principio de la función y antes de cualquier proposición ejecutable. Una *declaración* notifica las propiedades de una variable; consta de un nombre de tipo y una lista de variables, como

```
int fahr, celsius;
int lower, upper, step;
```

El tipo int significa que las variables de la lista son enteros, en contraste con float, que significa punto flotante, esto es, números que pueden tener una parte fraccionaria. El rango tanto de int como de float depende de la máquina que se está utilizando; los int de 16 bits, que están comprendidos entre el −32768 y +32767, son comunes, como lo son los int de 32 bits. Un número float típicamente es de 32 bits, por lo menos con seis dígitos significativos y una magnitud generalmente entre 10^{-38} y 10^{+38}.

Además de int y **float**, C proporciona varios tipos de datos básicos, incluyendo:

char	carácter —un solo byte
short	entero corto
long	entero largo
double	punto flotante de doble precisión

Los tamaños de estos objetos también dependen de la máquina. También existen *arreglos, estructuras* y *uniones* de estos tipos básicos, apuntadores a ellos y funciones que regresan valores con esos tipos, todo lo cual se verá en el momento oportuno.

Los cálculos en el programa de conversión de temperaturas principian con las *proposiciones de asignación*.

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

que asignan a las variables sus valores iniciales. Las proposiciones individuales se terminan con punto y coma.

Cada línea de la tabla se calcula de la misma manera por lo que se utiliza una iteración que se repite una vez por cada línea de salida; este es el propósito del ciclo **while**

```
while (fahr <= upper) {
    ...
}
```

El ciclo **while** funciona de la siguiente manera: se prueba la condición entre paréntesis. De ser verdadera (fahr es menor o igual que upper), el cuerpo del ciclo (las tres proposiciones entre llaves) se ejecuta. Luego la condición se prueba nuevamente, y si es verdadera, el cuerpo se ejecuta de nuevo. Cuando la prueba resulta falsa (fahr excede a upper) la iteración termina, y la ejecución continúa en la proposición que sigue al ciclo. No existe ninguna otra proposición en este programa, de modo que termina.

El cuerpo de un while puede tener una o más proposiciones encerradas entre llaves, como en el convertidor de temperaturas, o una sola proposición sin llaves, como en

```
while (i < j)
    i = 2 * i;
```


En cualquier caso, siempre se sangra la proposición controlada por el while con una tabulación (lo que se ha mostrado con cuatro espacios) para poder apreciar de un vistazo cuáles proposiciones están dentro del ciclo. El sangrado acentúa la estructura lógica del programa. Aunque a los compiladores de C no les importa la apariencia del programa, un sangrado y espaciamiento adecuados son muy importantes para hacer programas fáciles de leer. Se recomienda escribir una sola proposición por línea y utilizar espacios en blanco alrededor de los operadores para dar claridad al agrupamiento. La posición de las llaves es menos importante, aunque la gente mantiene creencias apasionadas. Se eligió uno de los varios estilos populares. Seleccione un estilo que le satisfaga y úselo en forma consistente.

La mayor parte del trabajo se realiza en el cuerpo del ciclo. La temperatura Celsius se calcula y se asigna a la variable **celsius** por la proposición.

```
celsius = 5 * (fahr-32) / 9;
```

La razón de multiplicar por 5 y después dividir entre 9 en lugar de solamente multiplicar por 5/9 es que en C, como en muchos otros lenguajes, la división de enteros *trunca* el resultado: cualquier parte fraccionaria se descarta. Puesto que 5 y 9 son enteros, 5/9 sería truncado a cero y así todas las temperaturas Celsius se reportarían como cero.

Este ejemplo también muestra un poco más acerca de cómo funciona printf. En realidad, printf es una función de propósito general para dar formato de salida, que se describirá con detalle en el capítulo 7. Su primer argumento es una cadena de caracteres que serán impresos, con cada % indicando en donde uno de los otros (segundo, tercero, ...) argumentos va a ser sustituido, y en qué forma será impreso. Por ejemplo, %d especifica un argumento entero, de modo que la proposición

```
printf("%d\t%d\n", fahr, celsius);
```

hace que los valores de los dos enteros **fahr** y **celsius** sean escritos, con una tabulación (\t) entre ellos.

Cada construcción % en el primer argumento de **printf** está asociada con el correspondiente segundo argumento, tercero, etc., y deben corresponder apropiadamente en número y tipo, o se tendrán soluciones incorrectas.

Con relación a esto, **printf** no es parte del lenguaje C; no existe propiamente una entrada o salida definida en C. **printf** es sólo una útil función de la biblioteca estándar de funciones que está accesible normalmente a los programas en C. Sin embargo, el comportamiento de printf está definido en el estándar ANSI, por lo que sus propiedades deben ser las mismas en cualquier compilador o biblioteca que se apegue a él.

Para concentrarnos en C, no hablaremos mucho acerca de la entrada y la salida hasta el capítulo 7. En particular, pospondremos el tema de la entrada con formato hasta entonces. Si se tiene que darle entrada a números, léase la discusión de la función **scanf** en la sección 7.4. La función **scanf** es como printf, exceptuando que lee de la entrada en lugar de escribir a la salida.

Existen un par de problemas con el programa de conversión de temperaturas. El más simple es que la salida no es muy estética debido a que los números no están justificados hacia su derecha. Esto es fácil de corregir; si aumentamos a cada %d de la proposición **printf** una amplitud, los números impresos serán justificados hacia su derecha dentro de sus campos. Por ejemplo, podría decirse

```
printf("%3d %6d\n", fahr, celsius);
```

para escribir el primer número de cada línea en un campo de tres dígitos de ancho, y el segundo en un campo de seis dígitos, como esto:

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
...
```

El problema más grave es que debido a que se ha utilizado aritmética de enteros, las temperaturas Celsius no son muy precisas; por ejemplo, 0°F es en realidad aproximadamente −17.8°C, no −17. Para obtener soluciones más precisas, se debe utilizar aritmética de punto flotante en lugar de entera. Esto requiere de algunos cambios en el programa. Aquí está una segunda versión:

```
# include <stdio.h>

/* imprime la tabla Fahrenheit-Celsius
     para fahr = 0, 20, ..., 300; versión de punto flotante */
main( )
{
     float fahr, celsius;
     int lower, upper, step;

     lower = 0;          /* límite superior de la tabla de temperaturas */
     upper = 300;        /* límite superior */
     step = 20;          /* tamaño del incremento */

     fahr = lower;
     while (fahr <= upper) {
          celsius = (5.0/9.0) * (fahr-32.0);
          printf("%3.0f %6.1f\n", fahr, celsius);
          fahr = fahr + step;
     }
}
```


Esto es muy semejante a lo anterior, excepto que **fahr** y **celsius** están declarados como **float**, y la fórmula de conversión está escrita en una forma más natural. No pudimos utilizar 5/9 en la versión anterior debido a que la división entera lo truncaría a cero. Sin embargo, un punto decimal en una constante indica que ésta es de punto flotante, por lo que 5.0/9.0 no se trunca debido a que es una relación de dos valores de punto flotante.

Si un operador aritmético tiene operandos enteros, se ejecuta una operación entera. Si un operador numérico tiene un operando de punto flotante y otro entero, este último será convertido a punto flotante antes de hacer la operación. Si se hubiera escrito **fahr − 32**, el **32** sería convertido automáticamente a punto flotante. Escribir constantes de punto flotante con puntos decimales explícitos, aun cuando tengan valores enteros, destaca su naturaleza de punto flotante para los lectores humanos.

Las reglas detalladas de cuándo los enteros se convierten a punto flotante se encuentran en el capítulo 2. Por ahora, nótese que la asignación

```
fahr = lower;
```

y la prueba

```
while (fahr <= upper)
```

también trabajan en la forma natural —el **int** se convierte a **float** antes de efectuarse la operación.

La especificación de conversión **%3.0f** del **printf** indica que se escribirá un número de punto flotante (en este caso **fahr**) por lo menos con tres caracteres de ancho, sin punto decimal y sin dígitos fraccionarios; %6.1f describe a otro número (**celsius**) que se escribirá en una amplitud de por lo menos 6 caracteres, con 1 dígito después del punto decimal. La salida se verá como sigue:

```
 0   -17.8
20    -6.7
40     4.4
...
```

La amplitud y la precisión pueden omitirse de una especificación: %6f indica que el número es por lo menos de seis caracteres de ancho; %.2f indica dos caracteres después del punto decimal, pero el ancho no está restringido; y %f únicamente indica escribir el número como punto flotante.

%d	escribe como entero decimal
%6d	escribe como entero decimal, por lo menos con 6 caracteres de amplitud
%f	escribe como punto flotante
%6f	escribe como punto flotante, por lo menos con 6 caracteres de amplitud

%.2f	escribe como punto flotante, con 2 caracteres después del punto decimal
%6.2f	escribe como punto flotante, por lo menos con 6 caracteres de ancho y 2 después del punto decimal

Entre otros, **printf** también reconoce %o para octal, %x para hexadecimal, %c para carácter, %s para cadena de caracteres y %% para % en sí.

Ejercicio 1-3. Modifique el programa de conversión de temperaturas de modo que escriba un encabezado sobre la tabla. □

Ejercicio 1-4. Escriba un programa que imprima la tabla correspondiente Celsius a Fahrenheit. □

1.3 La proposición for

Existen suficientes formas distintas de escribir un programa para una tarea en particular. Intentemos una variación del programa de conversión de temperaturas.

```
#include <stdio.h>

/* imprime la tabla Fahrenheit-Celsius */
main( )
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f/n", fahr, (5.0/9.0)*(fahr-32));
}
```

Este produce los mismos resultados, pero ciertamente se ve diferente. Un cambio importante es la eliminación de la mayoría de las variables; sólo permanece **fahr** y la hemos hecho int. Los límites inferior y superior y el tamaño del avance sólo aparecen como constantes dentro de la proposición **for**, que es una nueva construcción, y la expresión que calcula la temperatura Celsius ahora aparece como el tercer argumento de **printf** en vez de una proposición de asignación separada.

Este último cambio ejemplifica una regla general —en cualquier contexto en el que se permita utilizar el valor de una variable de algún tipo, es posible usar una expresión más complicada de ese tipo. Puesto que el tercer argumento de **printf** debe ser un valor de punto flotante para coincidir con %6.1f, cualquier expresión de punto flotante puede estar allí.

La proposición for es un ciclo, una forma generalizada del while. Si se compara con el while anterior, su operación debe ser clara. Dentro de los paréntesis existen tres secciones, separadas por punto y coma. La primera, la inicialización

```
fahr = 0
```


se ejecuta una vez, antes de entrar propiamente al ciclo. La segunda sección es la condición o prueba que controla el ciclo:

```
fahr <= 300
```

Esta condición se evalúa; si es verdadera, el cuerpo del ciclo (en este caso un simple printf) se ejecuta. Después el incremento de avance

```
fahr = fahr + 20
```

se ejecuta y la condición se vuelve a evaluar. El ciclo termina si la condición se hace falsa. Tal como con el **while**, el cuerpo del ciclo puede ser una proposición sencilla o un grupo de proposiciones encerradas entre llaves. La inicialización, la condición y el incremento pueden ser cualquier expresión.

La selección entre **while** y **for** es arbitraria, y se basa en aquello que parezca más claro. El **for** es por lo general apropiado para ciclos en los que la inicialización y el incremento son proposiciones sencillas y lógicamente relacionadas, puesto que es más compacto que el **while** y mantiene reunidas en un lugar a las proposiciones que controlan al ciclo.

Ejercicio 1-5. Modifique el programa de conversión de temperaturas de manera que escriba la tabla en orden inverso, esto es, desde 300 grados hasta 0. □

1.4 Constantes simbólicas

Una observación final antes de dejar definitivamente el tema de la conversión de temperaturas. Es una mala práctica poner "números mágicos" como 300 y 20 en un programa, ya que proporcionan muy poca información a quien tenga que leer el programa, y son difíciles de modificar en un forma sistemática. Una manera de tratar a esos números mágicos es darles nombres significativos. Una línea **#define** define un *nombre simbólico* o *constante simbólica* como una cadena de caracteres especial:

#define *nombre texto de reemplazo*

A partir de esto, cualquier ocurrencia de *nombre* (que no esté entre comillas ni como parte de otro nombre) se sustituirá por el *texto de reemplazo* correspondiente. El *nombre* tiene la misma forma que un nombre de variable: una secuencia de letras y dígitos que comienza con una letra. El *texto de reemplazo* puede ser cualquier secuencia de caracteres; no está limitado a números.

```
#include <stdio.h>

#define  LOWER  0      /* límite inferior de la tabla */
#define  UPPER  300    /* límite superior */
#define  STEP   20     /* tamaño del incremento */
```

```
/* imprime la tabla Fahrenheit-Celsius */
main( )
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Las cantidades **LOWER**, **UPPER** y **STEP** son constantes simbólicas, no variables, por lo que no aparecen entre las declaraciones. Los nombres de constantes simbólicas, por convención, se escriben con letras mayúsculas, de modo que se puedan distinguir fácilmente de los nombres de variables escritos con minúsculas. Nótese que no hay punto y coma al final de una línea **#define**.

1.5 Entrada y salida de caracteres

Ahora vamos a considerar una familia de programas relacionados para el procesamiento de datos de tipo carácter. Se encontrará que muchos programas sólo son versiones ampliadas de los prototipos que se tratan aquí.

El modelo de entrada y salida manejado por la biblioteca estándar es muy simple. La entrada y salida de texto, sin importar dónde fue originada o hacia dónde se dirige, se tratan como flujos (*streams*) de caracteres. Un *flujo de texto* es una secuencia de caracteres divididos entre líneas, cada una de las cuales consta de cero o más caracteres seguidos de un carácter nueva línea. La biblioteca es responsable de hacer que cada secuencia de entrada o salida esté de acuerdo con este modelo; el programador de C que utiliza la biblioteca no necesita preocuparse de cómo están representadas las líneas fuera del programa.

La biblioteca estándar proporciona varias funciones para leer o escribir un carácter a la vez, de las cuales **getchar** y **putchar** son las más simples. Cada vez que se invoca, **getchar** lee el *siguiente carácter de entrada* de una secuencia de texto y lo devuelve como su valor. Esto es, después de

```
c = getchar( )
```

la variable **c** contiene el siguiente carácter de entrada. Los caracteres provienen normalmente del teclado; la entrada de archivos se trata en el capítulo 7.

La función **putchar** escribe un carácter cada vez que se invoca:

```
putchar(c)
```

escribe el contenido de la variable entera **c** como un carácter, generalmente en la pantalla. Las llamadas a **putchar** y a **printf** pueden estar alternadas; la salida aparecerá en el orden en que se realicen las llamadas.

1.5.1 Copia de archivos

Con **getchar** y **putchar** se puede escribir una cantidad sorprendente de código útil sin saber nada más acerca de entrada y salida. El ejemplo más sencillo es un programa que copia la entrada en la salida, un carácter a la vez:

> *lee un carácter*
> while (*carácter no es indicador de fin de archivo*)
> *manda a la salida el carácter recién leído*
> *lee un carácter*

Al convertir esto en C se obtiene

```
#include <stdio.h>

/* copia la entrada a la salida; 1a. versión */
main( )
{
    int c;

    c = getchar( );
    while (c != EOF) {
        putchar(c);
        c = getchar( );
    }
}
```

El operador de relación **!=** significa "no igual a".

Lo que aparece como un carácter en el teclado o en la pantalla es, por supuesto, como cualquier otra cosa, almacenado internamente como un patrón de bits. El tipo **char** tiene la función específica de almacenar ese tipo de dato, pero también puede ser usado cualquier tipo de entero. Usamos **int** por una sutil pero importante razón.

El problema es distinguir el fin de la entrada de los datos válidos. La solución es que **getchar** devuelve un valor distintivo cuando no hay más a la entrada, un valor que no puede ser confundido con ningún otro carácter. Este valor se llama **EOF**, por "*end of file*" (fin de archivo)". Se debe declarar **c** con un tipo que sea lo suficientemente grande para almacenar cualquier valor que le regrese **getchar**. No se puede utilizar **char** puesto que **c** debe ser suficientemente grande como para mantener a **EOF** además de cualquier otro carácter. Por lo tanto, se emplea **int**.

EOF es un entero definido en <stdio.h>, pero el valor numérico específico no importa mientras que no sea el mismo que ningún valor tipo **char**. Utilizando la constante simbólica, hemos asegurado que nada en el programa depende del valor numérico específico.

El programa para copiar podría escribirse de modo más conciso por programadores experimentados de C. En lenguaje C, cualquier asignación, tal como

```
c = getchar( )
```

es una expresión y tiene un valor, el del lado izquierdo luego de la asignación. Esto significa que una asignación puede aparecer como parte de una expresión más larga. Si la asignación de un carácter a **c** se coloca dentro de la sección de prueba de un ciclo **while**, el programa que copia puede escribirse de la siguiente manera:

```
#include <stdio.h>

/* copia la entrada a la salida; 2a. versión */
main( )
{
    int c;

    while ((c = getchar( )) ! = EOF)
        putchar(c);
}
```

El **while** obtiene un carácter, lo asigna a **c**, y entonces prueba si el carácter fue la señal de fin de archivo. De no serlo, el cuerpo del **while** se ejecuta, escribiendo el carácter; luego se repite el **while**. Luego, cuando se alcanza el final de la entrada, el **while** termina y también lo hace **main**.

Esta versión centraliza la entrada —ahora hay sólo una referencia a **getchar**— y reduce el programa. El programa resultante es más compacto y más fácil de leer una vez que se domina el truco. Usted verá seguido este estilo. (Sin embargo, es posible descarriarse y crear código impenetrable, una tendencia que trataremos de reprimir.)

Los paréntesis que están alrededor de la asignación dentro de la condición son necesarios. La *precedencia* de **! =** es más alta que la de **=**, lo que significa que en ausencia de paréntesis la prueba de relación **!=** se realizaría antes de la asignación **=**. De esta manera, la proposición

```
c = getchar( ) != EOF
```

es equivalente a

```
c = (getchar( ) != EOF)
```

Esto tiene el efecto indeseable de hacer que **c** sea 0 o 1, dependiendo de si la llamada de **getchar** encontró fin de archivo. (En el capítulo 2 se trata este tema con más detalle).

Ejercicio 1-6. Verifique que la expresión **getchar ( ) != EOF** es 0 o 1. □

Ejercicio 1-7. Escriba un programa que imprima el valor de **EOF**. □

1.5.2 Conteo de caracteres

El siguiente programa cuenta caracteres y es semejante al programa que copia.

```
#include <stdio.h>

/ cuenta los caracteres de la entrada; 1a. versión */
main( )
{
    long nc;

    nc = 0;
    while (getchar( ) != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

La proposición

```
++nc;
```

presenta un nuevo operador, **++**, que significa ***incrementa en uno***. Es posible escribir **nc = nc + 1**, pero **++nc** es más conciso y muchas veces más eficiente. Hay un operador correspondiente -- para disminuir en 1. Los operadores **++** y **--** pueden ser tanto operadores prefijos (**++nc**) como postfijos (**nc++**); esas dos formas tienen diferentes valores dentro de las expresiones, como se demostrará en el capítulo 2, pero ambos **++nc** y **nc++** incrementan a **nc**. Por el momento adoptaremos la forma de prefijo.

El programa para contar caracteres acumula su cuenta en una variable **long** en lugar de una **int**. Los enteros **long** son por lo menos de 32 bits. Aunque en algunas máquinas **int** y **long** son del mismo tamaño, en otras un **int** es de 16 bits, con un valor máximo de 32767, y tomaría relativamente poca lectura a la entrada para desbordar un contador **int**. La especificación de conversión **%ld** indica a **printf** que el argumento correspondiente es un entero **long**.

Sería posible tener la capacidad de trabajar con números mayores empleando un **double** (float de doble precisión). También se utilizará una proposición **for** en lugar de un **while**, para demostrar otra forma de escribir el ciclo.

```
#include <stdio.h>

/* cuenta los caracteres de la entrada; 2a. versión */
main( )
{
    double nc;

    for (nc = 0; getchar( ) != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

printf utiliza %f tanto para float como para **double**; %.0f suprime la impresión del punto decimal y de la parte fraccionaria, que es cero.

El cuerpo de este ciclo **for** está vacío, debido a que todo el trabajo se realiza en las secciones de prueba e incremento. Pero las reglas gramaticales de C requieren que una proposición **for** tenga un cuerpo. El punto y coma aislado se llama *proposición nula*, y está aquí para satisfacer este requisito. Lo colocamos en una línea aparte para que sea visible.

Antes de abandonar el programa para contar caracteres, obsérvese que si la entrada no contiene caracteres, la prueba del **while** o del **for** no tiene éxito desde la primera llamada a **getchar**, y el programa produce cero, el resultado correcto. Esto es importante. Uno de los aspectos agradables acerca del **while** y del **for** es que hacen la prueba al inicio del ciclo, antes de proceder con el cuerpo. Si no hay nada que hacer, nada se hace, aun si ello significa no pasar a través del cuerpo del ciclo. Los programas deben actuar en forma inteligente cuando se les da una entrada de longitud cero. Las proposiciones **while** y **for** ayudan a asegurar que los programas realizan cosas razonables con condiciones de frontera.

1.5.3 Conteo de líneas

El siguiente programa cuenta líneas a la entrada. Como se mencionó anteriormente, la biblioteca estándar asegura que una secuencia de texto de entrada parezca una secuencia de líneas, cada una terminada por un carácter nueva línea. Por lo tanto, contar líneas es solamente contar caracteres nueva línea:

```
#include <stdio.h>

/* cuenta las líneas de la entrada */
main( )
{
    int c, nl;

    nl = 0;
    while ((c = getchar( )) != EOF)
        if (c == '/n')
            ++nl;
    printf("%d\n", nl);
}
```

El cuerpo del **while** consiste ahora en un **if**, el cual a su vez controla el incremento **++nl**. La proposición **if** prueba la condición que se encuentra entre paréntesis y, si la condición es verdadera, ejecuta la proposición (o grupo de proposiciones entre llaves) que le sigue. Hemos sangrado nuevamente para mostrar lo que controla cada elemento.

El doble signo de igualdad **==** es la notación de C para expresar "igual a" (como el = simple de Pascal o el .EQ. de Fortran). Este símbolo se emplea para distinguir la prueba de igualdad del = simple que utiliza C para la asignación. Un mensaje de alerta: los principiantes de C ocasionalmente escriben = cuando en realidad deben usar ==. Como se verá en el capítulo 2, el resultado es por lo general una expresión legal, de modo que no se obtendrá ninguna advertencia.

Un carácter escrito entre apóstrofos representa un valor entero igual al valor numérico del carácter en el conjunto de caracteres de la máquina. Esto se llama una *constante de carácter*, aunque sólo es otra forma de escribir un pequeño entero. Así, por ejemplo **'A'** es una constante de carácter; en el conjunto ASCII de caracteres su valor es 65, esto es, la representación interna del carácter **A**. Por supuesto **'A'** es preferible que 65: su significado es obvio, y es independiente de un conjunto de caracteres en particular.

Las secuencias de escape que se utilizan en constantes de cadena también son legales en constantes de carácter; así, **'\n'** significa el valor del carácter nueva línea, el cual es 10 en código ASCII. Se debe notar cuidadosamente que **'\n'** es un carácter simple, y en expresiones es sólo un entero; por otro lado, **"\n"** es una constante cadena que contiene sólo un carácter. En el capítulo 2 se trata el tema de cadenas versus caracteres.

Ejercicio 1-8. Escriba un programa que cuente espacios en blanco, tabuladores y nuevas líneas. □

Ejercicio 1.9. Escriba un programa que copie su entrada a la salida, reemplazando cada cadena de uno o más blancos por un solo blanco. □

Ejercicio 1-10. Escriba un programa que copie su entrada a la salida, reemplazando cada tabulación por \t, cada retroceso por \b y cada diagonal invertida por \\. Esto hace que las tabulaciones y los espacios sean visibles sin confusiones. □

1.5.4 Conteo de palabras

El cuarto en nuestra serie de programas útiles cuenta las líneas, palabras y caracteres, usando la definición de que una palabra es cualquier secuencia de caracteres que no contiene espacio en blanco ni tabulación ni nueva línea. Esta es una versión reducida del programa **wc** de UNIX.

```
#include <stdio.h>

#define IN 1      /* en una palabra */
#define OUT 0     /* fuera de una palabra */

/* cuenta líneas, palabras, y caracteres de la entrada */
main( )
{
    int c, nl, nw, nc, state;
```

```
    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar( )) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf ("%d %d %d\n", nl, nw, nc);
}
```

Cada vez que el programa encuentra el primer carácter de una palabra, contabiliza una palabra más. La variable state registra si actualmente el programa está o no sobre una palabra; al iniciar es "no está sobre una palabra", por lo que se asigna el valor OUT. Es preferible usar las constantes simbólicas IN y OUT que los valores literales 1 y 0, porque hacen el programa más legible. En un programa tan pequeño como éste, la diferencia es mínima, pero en programas más grandes el incremento en claridad bien vale el esfuerzo extra que se haya realizado para escribir de esta manera desde el principio. También se descubrirá que es más fácil hacer cambios extensivos en programas donde los números mágicos aparecen sólo como constantes simbólicas.

La línea

```
nl = nw = nc = 0;
```

inicializa a las tres variables en cero. Este no es un caso especial sino una consecuencia del hecho de que una asignación es una expresión con un valor, y que las asignaciones se asocian de derecha a izquierda. Es como si se hubiese escrito

```
nl = (nw = (nc = 0));
```

El operador || significa "O" (o bien "OR"), por lo que la línea

```
if (c == ' ' || c == '\n' || c == '\t')
```

dice "si c es un blanco *o* c es nueva línea *o* c es un tabulador". (Recuerde que la secuencia de escape \t es una representación visible del carácter tabulador.) Existe un correspondiente operador && para AND; su precedencia es más alta que la de ||. Las expresiones conectadas por && o || se evalúan de izquierda a derecha, y se garantiza que la evaluación terminará tan pronto como se conozca la verdad o falsedad. Si c es un blanco, no hay necesidad de probar si es una nueva línea o un tabulador, de modo que esas pruebas no se hacen. Esto no es de particular importancia en este caso, pero es significativo en situaciones más complicadas, como se verá más adelante.

El ejemplo muestra también un else, el cual especifica una acción alternativa si la condición de una proposición if es falsa. La forma general es

if (*expresión*)
  proposición$_1$
else
  proposición$_2$

Una y sólo una de las dos proposiciones asociadas con un if-else se realiza. Si la *expresión es verdadera*, se ejecuta *proposición*$_1$; si no lo es, se ejecuta *proposición*$_2$. Cada *proposición* puede ser una proposición sencilla o varias entre llaves. En el programa para contar palabras, la que está después del else es un if que controla dos proposiciones entre llaves.

Ejercicio 1-11. ¿Cómo probaría el programa para contar palabras? ¿Qué clase de entrada es la más conveniente para descubrir errores si éstos existen? □

Ejercicio 1-12. Escriba un programa que imprima su entrada una palabra por línea. □

1.6 Arreglos

Escribamos un programa para contar el número de ocurrencias de cada dígito, de caracteres espaciadores (blancos, tabuladores, nueva línea), y de todos los otros caracteres. Esto es artificioso, pero nos permite ilustrar varios aspectos de C en un programa.

Existen doce categorías de entrada, por lo que es conveniente utilizar un arreglo para mantener el número de ocurrencias de cada dígito, en lugar de tener diez variables individuales. Esta es una versión del programa:

```
#include <stdio.h>

/* cuenta dígitos, espacios blancos, y otros */
main( )
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;
```

```
        while ((c = getchar( )) != EOF)
            if (c >= '0' && c <= '9')
                ++ndigit[c-'0'];
            else if (c == ' ' || c == '\n' || c == '\t')
                ++nwhite;
            else
                ++nother;

        printf ("dígitos =");
        for (i = 0; i < 10; ++i)
            printf(" %d", ndigit[i]);
        printf(", espacios blancos = %d, otros = %d\n",
            nwhite, nother);
}
```

La salida de este programa al ejecutarlo sobre sí mismo es

```
dígitos = 9 3 0 0 0 0 0 0 0 1, espacios blancos = 123, otros = 345
```

La declaración

```
int ndigit [10];
```

declara **ndigit** como un arreglo de 10 enteros. En C, los subíndices de arreglos comienzan en cero, por lo que los elementos son **ndigit[0]**, **ndigit [1]**, ..., **ndigit[9]**. Esto se refleja en los ciclos **for** que inicializan e imprimen el arreglo.

Un subíndice puede ser cualquier expresión entera, lo que incluye a variables enteras como **i**, y constantes enteras.

Este programa en particular se basa en las propiedades de la representación de los dígitos como caracteres. Por ejemplo, la prueba

```
if (c >= '0' && c <= '9') ...
```

determina si el carácter en **c** es un dígito. Si lo es, el valor numérico del dígito es

```
c - '0'
```

Esto sólo funciona si '0', '1', ..., '9' tienen valores consecutivos ascendentes. Por fortuna, esto es así en todos los conjuntos de caracteres.

Por definición, los **char** son sólo pequeños enteros, por lo que las variables y las constantes **char** son idénticas a las **int** en expresiones aritméticas. Esto es natural y conveniente; por ejemplo, **c-'0'** es una expresión entera con un valor entre 0 y 9, correspondiente a los caracteres '0' a '9' almacenados en **c**, por lo que es un subíndice válido para el arreglo **ndigit**.

La decisión de si un carácter es dígito, espacio en blanco u otra cosa se realiza con la secuencia

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```


El patrón

> **if** (*condición*$_1$)
>   *proposición*$_1$
> **else if** (*condición*$_2$)
>   *proposición*$_2$
> ...
>   ...
> **else**
>   *proposición*$_n$

se encuentra frecuentemente en programas como una forma de expresar una decisión múltiple. Las *condiciones* se evalúan en orden desde el principio hasta que se satisface alguna *condición*; en ese punto se ejecuta la *proposición* correspondiente, y la construcción completa termina. (Cualquier *proposición* puede constar de varias proposiciones entre llaves.) Si no se satisface ninguna de las condiciones, se ejecuta la *proposición* que está después del **else** final, si ésta existe. Cuando se omiten el **else** y la *proposición* finales, tal como se hizo en el programa para contar palabras, no tiene lugar ninguna acción. Puede haber cualquier número de grupos de

> **else if** (*condición*)
>   *proposición*

entre el **if** inicial y el **else** final.

Se recomienda, por estilo, escribir ésta construcción tal como se ha mostrado; si cada **if** estuviese sangrado después del **else** anterior, una larga secuencia de decisiones podría rebasar el margen derecho de la página.

La proposición **switch**, que se tratará en el capítulo 3, proporciona otra forma de escribir una decisión múltiple, que es particularmente apropiada cuando la condición es determinar si alguna expresión entera o de carácter corresponde con algún miembro de un conjunto de constantes. Para contrastar, se presentará una versión de este programa, usando **switch**, en la sección 3.4.

Ejercicio 1-13. Escriba un programa que imprima el histograma de las longitudes de las palabras de su entrada. Es fácil dibujar el histograma con las barras horizontales; la orientación vertical es un reto más interesante. □

Ejercicio 1-14. Escriba un programa que imprima el histograma de las frecuencias con que se presentan diferentes caracteres leídos a la entrada. □

1.7 Funciones

En lenguaje C, una función es el equivalente a una subrutina o función en Fortran, o a un procedimiento o función en Pascal. Una función proporciona una forma conveniente de encapsular algunos cálculos, que se pueden emplear después sin preocuparse de su implantación. Con funciones diseñadas adecuadamente, es posible ignorar *cómo* se realiza un trabajo; es suficiente saber *qué* se hace. El lenguaje C hace que el uso de funciones sea fácil, conveniente y eficiente; es común ver una función corta definida y empleada una sola vez, únicamente porque eso esclarece alguna parte del código.

Hasta ahora sólo se han utilizado funciones como **printf**, **getchar** y **putchar**, que nos han sido proporcionadas; ya es el momento de escribir unas pocas nosotros mismos. Dado que C no posee un operador de exponenciación como el ** de Fortran, ilustremos el mecanismo de la definición de una función al escribir la función **power(m,n)**, que eleva un entero **m** a una potencia entera y positiva **n**. Esto es, el valor de **power(2,5)** es 32. Esta función no es una rutina de exponenciación práctica, puesto que sólo maneja potencias positivas de enteros pequeños, pero es suficiente para ilustración (la biblioteca estándar contiene una función **pow(x,y)** que calcula x^y).

A continuación se presenta la función **power** y un programa **main** para utilizarla, de modo que se vea la estructura completa de una vez.

```
#include <stdio.h>

int power(int m, int n);

/* prueba la función power */
main( )
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power:  eleva la base a la n-ésima potencia; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```


Una definición de función tiene la forma siguiente:

> *tipo-de-retorno nombre-de-función* (*declaración de parámetros, si los hay*)
> {
>   *declaraciones*
>   *proposiciones*
> }

Las definiciones de función pueden aparecer en cualquier orden y en uno o varios archivos fuente, pero una función no puede separarse en archivos diferentes. Si el programa fuente aparece en varios archivos, tal vez se tengan que especificar más cosas al compilar y cargarlo que si estuviera en uno solo, pero eso es cosa del sistema operativo, no un atributo del lenguaje. Por ahora supondremos que ambas funciones están en el mismo archivo y cualquier cosa que se haya aprendido acerca de cómo ejecutar programas en C, aún funcionarán.

La función **power** se invoca dos veces por **main**, en la línea

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Cada llamada pasa dos argumentos a **power**, que cada vez regresa un entero, al que se pone formato y se imprime. En una expresión, **power(2,i)** es un entero tal como lo son **2** e **i**. (No todas las funciones producen un valor entero; lo que se verá en el capítulo 4.)

La primera línea de la función **power**,

```
int power(in base, int n)
```

declara los tipos y nombres de los parámetros, así como el tipo de resultado que la función devuelve. Los nombres que emplea **power** para sus parámetros son locales a la función y son invisibles a cualquier otra función: otras rutinas pueden utilizar los mismos nombres sin que exista problema alguno. Esto también es cierto para las variables **i** y **p**: la **i** de **power** no tiene nada que ver con la **i** de **main**.

Generalmente usaremos *parámetro* para una variable nombrada en la lista entre paréntesis de la definición de una función, y *argumento* para el valor empleado al hacer la llamada de la función. Los términos *argumento formal* y *argumento real* se emplean en ocasiones para hacer la misma distinción.

El valor que calcula **power** se regresa a **main** por medio de la proposición **return**, a la cual le puede seguir cualquier expresión:

> return *expresión*

Una función no necesita regresar un valor; una proposición **return** sin expresión hace que el control regrese al programa, pero no devuelve algún valor de utilidad, como se haría al "caer al final" de una función al alcanzar la llave derecha de terminación. Además, la función que llama puede ignorar el valor que regresa una función.

Probablemente haya notado que hay una proposición **return** al final de **main**. Puesto que **main** es una función como cualquier otra, también puede regresar un valor a quien la invoca, que es en efecto el medio ambiente en el que el programa

se ejecuta. Típicamente, un valor de regreso cero implica una terminación normal; los valores diferentes de cero indican condiciones de terminación no comunes o erróneas. Para buscar la simplicidad, se han omitido hasta ahora las proposiciones **return** de las funciones **main**, pero se incluirán más adelante, como un recordatorio de que los programas deben regresar su estado final a su medio ambiente.

La declaración

```
int power(int m, int n);
```

precisamente antes de **main**, indica que **power** es una función que espera dos argumentos **int** y regresa un **int**. Esta declaración, a la cual se le llama *función prototipo*, debe coincidir con la definición y uso de **power**. Es un error el que la definición de una función o cualquier uso que de ella se haga no corresponda con su prototipo.

Los nombres de los parámetros no necesitan coincidir; de hecho, son optativos en el prototipo de una función, de modo que para el prototipo se pudo haber escrito

```
int power(int, int);
```

No obstante, unos nombres bien seleccionados son una buena documentación, por lo que se emplearán frecuentemente.

Una nota histórica: La mayor modificación entre ANSI C y las versiones anteriores es cómo están declaradas y definidas las funciones. En la definición original de C, la función **power** se pudo haber escrito de la siguiente manera:

```
/* power: eleva la base a n-ésima potencia; n >= 0 */
/*        (versión en estilo antiguo) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Los parámetros se nombran entre los paréntesis y sus tipos se declaran antes de abrir la llave izquierda; los parámetros que no se declaran se toman como **int**. (El cuerpo de la función es igual a la anterior.)

La declaración de **power** al inicio del programa pudo haberse visto como sigue:

```
int power( );
```

No se permitió ninguna lista de parámetros, de modo que el compilador no pudo revisar con facilidad que **power** fuera llamada correctamente. De hecho, puesto que por omisión se podía suponer que **power** regresaba un entero, toda la declaración podría haberse omitido.

La nueva sintaxis de los prototipos de funciones permite que sea mucho más fácil para el compilador detectar errores en el número o tipo de argumentos. El viejo estilo de declaración y definición aún funciona en ANSI C, al menos por un periodo de transición, pero se recomienda ampliamente que se utilice la nueva forma si se tiene un compilador que la maneje.

Ejercicio 1-15. Escriba de nuevo el programa de conversión de temperatura de la sección 1.2, de modo que utilice una función para la conversión. □

1.8 Argumentos—llamadas por valor

Hay un aspecto de las funciones de C que puede parecer poco familiar a los programadores acostumbrados a otros lenguajes, particularmente Fortran. En C, todos los argumentos de una función se pasan "por valor". Esto significa que la función que se invoca recibe los valores de sus argumentos en variables temporales y no en las originales. Esto conduce a algunas propiedades diferentes a las que se ven en lenguajes con "llamadas por referencia" como Fortran o con parámetros **var** en Pascal, en donde la rutina que se invoca tiene acceso al argumento original, no a una copia local.

La diferencia principal es que en C la función que se invoca no puede alterar directamente una variable de la función que hace la llamada; sólo puede modificar su copia privada y temporal.

Sin embargo, la llamada por valor es una ventaja, no una desventaja. Por lo común, esto conduce a elaborar programas más compactos con pocas variables extrañas, debido a que los parámetros se tratan en la función invocada como variables locales convenientemente inicializadas. Por ejemplo, he aquí una versión de **power** que utiliza esta propiedad.

```
/* power: eleva la base a la n-ésima potencia; n>=0; versión 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

El parámetro **n** se utiliza como una variable temporal, y se decrementa (un ciclo **for** que se ejecuta hacia atrás) hasta que llega a cero; ya no es necesaria la variable **i**. Cualquier cosa que se le haga a **n** dentro de **power** no tiene efecto sobre el argumento con el que se llamó originalmente **power**.

Cuando sea necesario, es posible hacer que una función modifique una variable dentro de una rutina invocada. La función que llama debe proporcionar la *dirección* de la variable que será cambiada (técnicamente un *apuntador* a la variable), y la función que se invoca debe declarar que el parámetro sea un apuntador y tenga acceso a la variable indirectamente a través de él. Los apuntadores se tratarán en el capítulo 5.

La historia es diferente con los arreglos. Cuando el nombre de un arreglo se emplea como argumento, el valor que se pasa a la función es la localización o la dirección del principio del arreglo —no hay copia de los elementos del arreglo. Al colocarle subíndices a este valor, la función puede tener acceso y alterar cualquier elemento del arreglo. Este es el tema de la siguiente sección.

1.9 Arreglos de caracteres

El tipo de arreglo más común en C es el de caracteres. Para ilustrar el uso de arreglos de caracteres y funciones que los manipulan, escriba un programa que lea un conjunto de líneas de texto e imprima la de mayor longitud. El pseudocódigo es bastante simple:

```
while (hay otra línea)
    if (es más larga que la anterior más larga)
        guárdala
        guarda su longitud
imprime la línea más larga
```

Este pseudocódigo deja en claro que el programa se divide naturalmente en partes. Una trae una nueva línea, otra la prueba y el resto controla el proceso.

Puesto que la división de las partes es muy fina, lo correcto será escribirlas de ese modo. Así pues, escribamos primero una función **getline** para extraer la siguiente línea de la entrada. Trataremos de hacer a la función útil en otros contextos. Al menos, **getline** tiene que regresar una señal acerca de la posibilidad de un fin de archivo; un diseño de más utilidad deberá retornar la longitud de la línea, o cero si se encuentra el fin de archivo. Cero es un regreso de fin de archivo aceptable debido a que nunca es una longitud de línea válida. Cada línea de texto tiene al menos un carácter; incluso una línea que sólo contenga un carácter nueva línea tiene longitud 1.

Cuando se encuentre una línea que es mayor que la anteriormente más larga, se debe guardar en algún lugar. Esto sugiere una segunda función **copy**, para copiar la nueva línea a un lugar seguro.

Finalmente, se necesita un programa principal para controlar **getline** y **copy**. El resultado es el siguiente:


```
#include <stdio.h>
#define MAXLINE 1000    /* tamaño máximo de la línea de entrada */
int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* imprime la línea de entrada más larga */
main( )
{
    int len;                /* longitud actual de la línea */
    int max;                /* máxima longitud vista hasta el momento */
    char line[MAXLINE];     /* línea de entrada actual */
    char longest[MAXLINE];  /* la línea más larga se guarda aquí */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0)        /* hubo una línea */
        printf("%s", longest);
    return 0;
}

/* getline: lee una línea en s, regresa su longitud */
int getline(char s[], int lim)
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar( )) !=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}
/* copy: copia 'from' en 'to'; supone que to es suficientemente grande */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Las funciones **getline** y **copy** están declaradas al principio del programa, que se supone está contenido en un archivo.

main y **getline** se comunican a través de un par de argumentos y un valor de retorno. En **getline** los argumentos se declaran por la línea

```
int getline(char s[], int lim)
```

que especifica que el primer argumento, **s**, es un arreglo, y el segundo, **lim**, es un entero. El propósito de proporcionar el tamaño de un arreglo es fijar espacio de almacenamiento contiguo. La longitud del arreglo **s** no es necesaria en **getline**, puesto que su tamaño se fija en **main**. En **getline** se utiliza **return** para regresar un valor a quién lo llama, tal como hizo la función **power**. Esta línea también declara que **getline** regresa un **int**; puesto que **int** es el valor de retorno por omisión, puede suprimirse.

Algunas funciones regresan un valor útil; otras, como **copy**, se emplean únicamente por su efecto y no regresan un valor. El tipo de retorno de **copy** es **void**, el cual establece explícitamente que ningún valor se regresa.

En **getline** se coloca el carácter '\0' (*carácter nulo*, cuyo valor es cero) al final del arreglo que está creando, para marcar el fin de la cadena de caracteres. Esta convención también se utiliza por el lenguaje C: cuando una constante de carácter como

```
"hola\n"
```

aparece en un programa en C, se almacena como un arreglo que contiene los caracteres de la cadena y termina con un '\0' para marcar el fin.

h	o	l	a	\n	\0

La especificación de formato **%s** dentro de **printf** espera que el argumento correspondiente sea una cadena representada de este modo; **copy** también se basa en el hecho de que su argumento de entrada se termina con '\0', y copia este carácter dentro del argumento de salida. (Todo esto implica que '\0', no es parte de un texto normal.)

Es útil mencionar de paso que aun un programa tan pequeño como éste presenta algunos problemas de diseño. Por ejemplo, ¿qué debe hacer **main** si encuentra una línea que es mayor que su límite? **getline** trabaja en forma segura, en ese caso detiene la recopilación cuando el arreglo está lleno, aunque no encuentre el carácter nueva línea. Probando la longitud y el último carácter devuelto, **main** puede determinar si la línea fue demasiado larga, y entonces realiza el tratamiento que se desee. Por brevedad, hemos ignorado el asunto.

Para un usuario de **getline** no existe forma de saber con anticipación cuán larga podrá ser una línea de entrada, por lo que **getline** revisa un posible desbordamiento (*overflow*). Por otro lado, el usuario de **copy** ya conoce (o lo puede averiguar) cuál es el tamaño de la cadena, por lo que decidimos no agregar comprobación de errores en ella.

Ejercicio 1-16. Corrija la rutina principal del programa de la línea más larga de modo que imprima correctamente la longitud de líneas de entrada arbitrariamente largas, y tanto texto como sea posible. □

Ejercicio 1-17. Escriba un programa que imprima todas las líneas de entrada que sean mayores de 80 caracteres. □

Ejercicio 1-18. Escriba un programa que elimine los blancos y los tabuladores que estén al final de cada línea de entrada, y que borre completamente las líneas en blanco. □

Ejercicio 1-19. Escriba una función **reverse(s)** que invierta la cadena de caracteres **s**. Usela para escribir un programa que invierta su entrada, línea a línea. □

1.10 Variables externas y alcance

Las variables que están en **main**, tal como **line**, **longest**, etc., son privadas o locales a ella. Debido a que son declaradas dentro de **main**, ninguna otra función puede tener acceso directo a ellas. Lo mismo también es válido para variables de otras funciones; por ejemplo, la variable **i** en **getline** no tiene relación con la **i** que está en **copy**. Cada variable local de una función comienza a existir sólo cuando se llama a la función, y desaparece cuando la función termina. Esto es por lo que tales variables son conocidas como variables *automáticas*, siguiendo la terminología de otros lenguajes. Aquí se utilizará en adelante el término automático para hacer referencia a esas variables locales. (En el capítulo 4 se discute la categoría de almacenamiento **estática**, en la que las variables locales sí conservan sus valores entre llamadas.)

Debido a que las variables locales aparecen y desaparecen con la invocación de funciones, no retienen sus valores entre dos llamadas sucesivas, y deben ser inicializadas explícitamente en cada entrada. De no hacerlo, contendrán "basura".

Como una alternativa a las variables automáticas, es posible definir variables que son *externas* a todas las funciones, esto es, variables a las que toda función puede tener acceso por su nombre. (Este mecanismo es parecido al COMMON de Fortran o a las variables de Pascal declaradas en el bloque más exterior.) Debido a que es posible tener acceso global a las variables externas, éstas pueden ser usadas en lugar de listas de argumentos para comunicar datos entre funciones. Además, puesto que las variables externas se mantienen permanentemente en existencia, en lugar de aparecer y desaparecer cuando se llaman y terminan las funciones, mantienen sus valores aun después de que regresa la función que los fijó.

Una variable externa debe *definirse*, exactamente una vez, fuera de cualquier función; esto fija un espacio de almacenamiento para ella. La variable también debe *declararse* en cada función que desee tener acceso a ella; esto establece el tipo de la variable. La declaración debe ser una proposición **extern** explícita, o bien puede estar implícita en el contexto. Para concretar la discusión, reescribamos el

programa de la línea más larga con **line**, **longest** y **max** como variables externas. Esto requiere cambiar las llamadas, declaraciones y cuerpos de las tres funciones.

```
#include <stdio.h>

#define MAXLINE 1000        /* máximo tamaño de una línea de entrada */

int max;                    /* máxima longitud vista hasta el momento */
char line[MAXLINE];         /* línea de entrada actual */
char longest[MAXLINE];      /* la línea más larga se guarda aquí */

int getline(void);
void copy(void);


/* imprime la línea de entrada más larga; versión especializada */
main( )
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline( )) > 0)
        if (len > max) {
            max = len;
            copy( );
        }
    if (max > 0)        /* hubo una línea */
        printf("%s", longest);
    return 0;
}


/* getline: versión especializada */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE-1
        && (c=getchar( )) != EOF && c != '\n'; ++i)
            line[i] = c;
```



```
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}

/* copy: versión especializada */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Las variables externas de **main**, **getline** y **copy** están definidas en las primeras líneas del ejemplo anterior, lo que establece su tipo y causa que se les asigne espacio de almacenamiento. Desde el punto de vista sintáctico, las definiciones externas son exactamente como las definiciones de variables locales, pero puesto que ocurren fuera de las funciones, las variables son externas. Antes de que una función pueda usar una variable externa, se debe hacer saber el nombre de la variable a la función. Una forma de hacer esto es escribir una declaración **extern** dentro de la función; la declaración es la misma que antes, excepto por la palabra reservada **extern**.

Bajo ciertas circunstancias, la declaración **extern** se puede omitir. Si la definición de una variable externa ocurre dentro del archivo fuente antes de su uso por una función en particular, entonces no es necesario el uso de una declaración **extern** dentro de la función. La declaración **extern** en **main**, **getline** y **copy** es, por lo tanto, redundante. De hecho, una práctica común, es poner las definiciones de todas las variables externas al principio del archivo fuente y después omitir todas las declaraciones **extern**.

Si el programa está en varios archivos fuente y una variable se define en *archivo1* y se utiliza en *archivo2* y *archivo3*, entonces se necesitan declaraciones **extern** en *archivo2* y *archivo3* para conectar las ocurrencias de la variable. La práctica común es reunir declaraciones **extern** de variables y funciones en un archivo separado, llamado históricamente *header*, que es incluido por #include al principio de cada archivo fuente. El sufijo .h se usa por convención para nombres de headers. Las funciones de la biblioteca estándar, por ejemplo, están declaradas en *headers* como **<stdio.h>**. Este tema se trata ampliamente en el capítulo 4, y la biblioteca en el capítulo 7 y en el apéndice B.

Puesto que las versiones especializadas de **getline** y **copy** no tienen argumentos, la lógica sugeriría que sus prototipos al principio del archivo deben ser **getline( )** y **copy( )**. Pero por compatibilidad con programas de C anteriores, el estándar toma a una lista vacía como una declaración al viejo estilo, y suspende toda revisión de listas de argumentos; para una lista explícitamente vacía debe emplearse la palabra **void**. Esto se discutirá en el capítulo 4.

Se debe notar que empleamos cuidadosamente las palabras *definición* y *declaración* cuando nos referimos a variables externas en esta sección. La palabra "definición" se refiere al lugar donde se crea la variable o se le asigna un lugar de almacenamiento; "declaración" se refiere al lugar donde se establece la naturaleza de la variable pero no se le asigna espacio.

A propósito, existe una tendencia a convertir todo en variables **extern**, debido a que aparentemente simplifica las comunicaciones —las listas de argumentos son cortas y las variables existen siempre, cuando se les requiere. Pero las variables externas existen siempre, aun cuando no hacen falta. Descansar fuertemente sobre variables externas es peligroso, puesto que lleva a programas cuyas conexiones entre datos no son completamente obvias —las variables pueden cambiarse en forma inesperada e inadvertida, y el programa es difícil de modificar. La segunda versión del programa de la línea mayor es inferior a la primera, en parte por las anteriores razones y en parte porque destruye la generalidad de dos útiles funciones, introduciendo en ellas los nombres de las variables que manipula.

Hasta este punto hemos descrito lo que podría llamarse los fundamentos convencionales de C. Con estos fundamentos, es posible escribir programas útiles de tamaño considerable, y probablemente sería una buena idea hacer una pausa suficientemente grande para realizarlos. Estos ejercicios sugieren programas de complejidad algo mayor que los anteriores del capítulo.

Ejercicio 1-20. Escriba un programa **detab** que reemplace tabuladores de la entrada con el número apropiado de blancos para espaciar hasta el siguiente paro de tabulación. Considere un conjunto fijo de paros de tabulación, digamos cada *n* columnas. ¿Debe ser *n* una variable o un parámetro simbólico? □

Ejercicio 1-21. Escriba un programa **entab** que reemplace cadenas de blancos por el mínimo número de tabuladores y blancos para obtener el mismo espaciado. Considere los paros de tabulación de igual manera que para **detab**. Cuando un tabulador o un simple espacio en blanco fuese suficiente para alcanzar un paro de tabulación, ¿a cuál se le debe dar preferencia? □

Ejercicio 1-22. Escriba un programa para "doblar" líneas grandes de entrada en dos o más líneas más cortas después del último carácter no blanco que ocurra antes de la *n*-ésima columna de entrada. Asegúrese de que su programa se comporte apropiadamente con líneas muy largas, y de que no hay blancos o tabuladores antes de la columna especificada. □

Ejercicio 1-23. Escriba un programa para eliminar todos los comentarios de un programa en C. No olvide manejar apropiadamente las cadenas entre comillas y las constantes de carácter. Los comentarios de C no se anidan. □

Ejercicio 1-24. Escriba un programa para revisar los errores de sintaxis rudimentarios de un programa en C, como paréntesis, llaves y corchetes no alineados. No olvide las comillas ni los apóstrofos, las secuencias de escape y los comentarios. (Este programa es difícil si se hace completamente general.) □

CAPITULO 2: Tipos, operadores y expresiones

Las variables y las constantes son los objetos de datos básicos que se manipulan en un programa. Las declaraciones muestran las variables que se van a utilizar y establecen el tipo que tienen y algunas veces cuáles son sus valores iniciales. Los operadores especifican lo que se hará con las variables. Las expresiones combinan variables y constantes para producir nuevos valores. El tipo de un objeto determina el conjunto de valores que puede tener y qué operaciones se pueden realizar sobre él. Estos son los temas de este capítulo.

El estándar ANSI ha hecho muchos pequeños cambios y agregados a los tipos básicos y a las expresiones. Ahora hay formas **signed** y **unsigned** de todos los tipos enteros, y notaciones para constantes sin signo y constantes de carácter hexadecimales. Las operaciones de punto flotante pueden hacerse en precisión sencilla; también hay un tipo **long double** para precisión extendida. Las constantes de cadena pueden concatenarse al tiempo de compilación. Las enumeraciones son ya parte del lenguaje, formalizando una característica pendiente por mucho tiempo. Los objetos pueden ser declarados **const**, lo que impide que cambien. Las reglas para conversión automática entre tipos aritméticos se aumentaron para manejar el ahora más rico conjunto de tipos.

2.1 Nombres de variables

Aunque no lo mencionamos en el capítulo 1, existen algunas restricciones en los nombre de las variables y de las constantes simbólicas. Los nombres se componen de letras y dígitos; el primer carácter debe ser una letra. El carácter de subrayado "_" cuenta como una letra; algunas veces es útil para mejorar la legibilidad de nombres largos de variables. Sin embargo, no se debe comenzar los nombres de variables con este carácter, puesto que las rutinas de biblioteca con frecuencia usan tales nombres. Las letras mayúsculas y minúsculas son distintas, de tal manera que **x** y **X** son dos nombres diferentes. La práctica tradicional de C es usar letras minúsculas para nombres de variables, y todo en mayúsculas para constantes simbólicas.

Al menos los primeros 31 caracteres de un nombre interno son significativos. Para nombres de funciones y variables externas el número puede ser menor que 31, puesto que los nombres externos los pueden usar los ensambladores y los cargadores, sobre los que el lenguaje no tiene control. Para nombres externos, el estándar garantiza distinguir sólo para 6 caracteres y sin diferenciar mayúsculas de minúsculas. Las palabras clave como **if**, **else**, **int**, **float**, etc., están reservadas: no se pueden utilizar como nombres de variables. Todas ellas deben escribirse con minúsculas.

Es conveniente elegir nombres que estén relacionados con el propósito de la variable, que no sea probable confundirlos tipográficamente. Nosotros tendemos a utilizar nombres cortos para variables locales, especialmente índices de iteraciones, y nombres más largos para variables externas.

2.2 Tipos y tamaños de datos

Hay unos cuantos tipos de datos básicos en C:

char	un solo byte, capaz de contener un carácter del conjunto de caracteres local.
int	un entero, normalmente del tamaño natural de los enteros en la máquina en la que se ejecuta.
float	punto flotante de precisión normal.
double	punto flotante de doble precisión.

Además, existen algunos calificadores que se aplican a estos tipos básicos. **short** y **long** se aplican a enteros:

```
short int sh;
long int counter;
```

La palabra **int** puede omitirse de tales declaraciones, lo que típicamente se hace.

La intención es que **short** y **long** puedan proporcionar diferentes longitudes de enteros donde sea práctico; **int** será normalmente el tamaño natural para una máquina en particular. A menudo **short** es de 16 bits y **long** de 32; **int** es de 16 o de 32 bits. Cada compilador puede seleccionar libremente los tamaños apropiados para su propio hardware, sujeto sólo a la restricción de que los **short**s e **int**s son, por lo menos de 16 bits, los **long**s son por lo menos de 32 bits y el **short** no es mayor que **int**, el cual a su vez no es mayor que **long**.

El calificador **signed** o **unsigned** puede aplicarse a **char** o a cualquier entero. Los números **unsigned** son siempre positivos o cero y obedecen las leyes de la aritmética módulo 2^n, donde n es el número de bits en el tipo. Así, por ejemplo, si los **char** son de 8 bits, las variables **unsigned char** tienen valores entre 0 y 255, en tanto que las variables **signed char** tienen valores entre −128 y 127 (en una máquina de complemento a dos). El hecho de que los **char**s ordinarios sean con signo o sin él depende de la máquina, pero los caracteres que se pueden imprimir son siempre positivos.

El tipo **long double** especifica punto flotante de precisión extendida. Igual que con los enteros, los tamaños de objetos de punto flotante se definen en la implantación; **float**, **double** y **long double** pueden representar uno, dos o tres tamaños distintos.

Los archivos de encabezado *headers* estándar **<limits.h>** y **<float.h>** contienen constantes simbólicas para todos esos tamaños, junto con otras propiedades de la máquina y del compilador, los cuales se discuten en el apéndice B.

Ejercicio 2-1. Escriba un programa para determinar los rangos de variables **char**, **short**, **int** y **long**, tanto **signed** como **unsigned**, imprimiendo los valores apropiados de los *headers* estándar y por cálculo directo. Es más difícil si los calcula: determine los rangos de los varios tipos de punto flotante. □

2.3 Constantes

Una constante entera como **1234** es un **int**. Una constante **long** se escribe con una **l** (ele) o **L** terminal, como en **123456789L**; un entero demasiado grande para caber dentro de un **int** también será tomado como **long**. Las constantes sin signo se escriben con una **u** o **U**, terminal y el sufijo **ul** o **UL** indica **unsigned long**.

Las constantes de punto flotante contienen un punto decimal (**123.4**) o un exponente (**1e−2**) o ambos; su tipo es **double**, a menos que tengan sufijo. Los sufijos **f** o **F** indican una constante **float**; **l** o **L** indican un **long double**.

El valor de un entero puede especificarse en forma octal o hexadecimal en lugar de decimal. Un **0** (cero) al principio en una constante entera significa octal; **0x** ó **0X** al principio significa hexadecimal. Por ejemplo, el decimal 31 puede escribirse como **037** en octal y **0x1f** ó **OX1F** en hexadecimal. Las constantes octales y hexadecimales también pueden ser seguidas por **L** para convertirlas en **long** y **U** para hacerlas **unsigned**: **OXFUL** es una constante **unsigned long** con valor de 15 en decimal.

Una *constante de carácter* es un entero, escrito como un carácter dentro de apóstrofos, tal como **'x'**. El valor de una constante de carácter es el valor numérico del carácter en el conjunto de caracteres de la máquina. Por ejemplo, en el conjunto de caracteres ASCII el carácter constante **'0'** tiene el valor de 48, el cual no está relacionado con el valor numérico 0. Si escribimos **'0'** en vez de un valor numérico como 48 que depende del conjunto de caracteres, el programa es independiente del valor particular y más fácil de leer. Las constantes de carácter participan en operaciones numéricas tal como cualesquier otros enteros, aunque se utilizan más comúnmente en comparaciones con otros caracteres.

Ciertos caracteres pueden ser representados en constante de carácter y de cadena, por medio de secuencias de escape como **\n** (nueva línea); esas secuencias se ven

como dos caracteres, pero representan sólo uno. Además, un patrón de bits arbitrario de tamaño de un byte puede ser especificado por

'*ooo*'

en donde *ooo* son de uno a tres dígitos octales (0...7) o por

'*xhh*'

en donde *hh* son uno o más dígitos hexadecimales (0...9, a...f, A...F). Así podríamos escribir

```
#define VTAB '\013'    /* tab vertical ASCII */
#define BELL '\007'    /* carácter campana ASCII */
```

o, en hexadecimal,

```
#define VTAB '\xb'     /* tab vertical ASCII */
#define BELL '\x7'     /* carácter campana ASCII */
```

El conjunto completo de secuencias de escape es

\a	carácter de alarma (campana)	\\	diagonal invertida
\b	retroceso	\?	interrogación
\f	avance de hoja	\'	apóstrofo
\n	nueva línea	\"	comillas
\r	regreso de carro	\ooo	número octal
\t	tabulador horizontal	\xhh	número hexadecimal
\v	tabulador vertical		

La constante de carácter '\0' representa el carácter con valor cero, el carácter nulo. '\0' a menudo se escribe en vez de 0 para enfatizar la naturaleza de carácter de algunas expresiones, pero el valor numérico es precisamente 0.

Una *expresión constante* es una expresión que sólo inmiscuye constantes. Tales expresiones pueden ser evaluadas durante la compilación en vez de que se haga en tiempo de ejecución, y por tanto pueden ser utilizadas en cualquier lugar en que pueda encontrarse una constante, como en

```
#define MAXLINE 1000
char line [MAXLINE + 1];
```

o

```
#define LEAP 1   /* en años bisiestos*/
int days [31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Una *constante de cadena* o *cadena literal*, es una secuencia de cero o más caracteres encerrados entre comillas, como en

"Soy una cadena"

o

""/* la cadena vacía */

Las comillas no son parte de la cadena, sólo sirven para delimitarla. Las mismas secuencias de escape utilizadas en constantes de carácter se aplican en cadenas; \" representa el carácter comillas. Las constantes de cadena pueden ser concatenadas en tiempo de compilación:

"hola," "mundo"

es equivalente a

"hola, mundo"

Esto es útil para separar cadenas largas entre varias líneas fuente.

Técnicamente, una constante de cadena es un arreglo de caracteres. La representación interna de una cadena tiene un carácter nulo '\0' al final, de modo que el almacenamiento físico requerido es uno más del número de caracteres escritos entre las comillas. Esta representación significa que no hay límite en cuanto a qué tan larga puede ser una cadena, pero los programas deben leer completamente una cadena para determinar su longitud. La función **strlen** (s) de la biblioteca estándar regresa la longitud de su argumento s de tipo cadena de caracteres, excluyendo el '\0' terminal. Aquí está nuestra versión:

```
/* strlen: regresa la longitud de s */
int strlen(char s[ ])
{
    int i;

    i = 0;
    while (s[i] ! = '\0')
        + + i;
    return i;
}
```

strlen y otras funciones para cadenas están declaradas en el *header* estándar <string.h>.

Se debe ser cuidadoso al distinguir entre una constante de carácter y una cadena que contiene un sólo carácter: 'x' no es lo mismo que "x". El primero es un entero, utilizado para producir el valor numérico de la letra *x* en el conjunto de caracteres de la máquina. El último es un arreglo de caracteres que contiene un carácter (la letra *x*) y un '\0'.

Existe otra clase de constante, la *constante de enumeración*. Una enumeración es una lista de valores enteros constantes, como en

```
enum boolean {NO, YES};
```

El primer nombre en un **enum** tiene valor 0, el siguiente 1, y así sucesivamente, a menos que sean especificados valores explícitos. Si no todos los valores son es-

pecificados, los valores no especificados continúan la progresión a partir del último valor que sí lo fue, como en el segundo de esos ejemplos:

```
enum escapes { BELL = '\a', RETROCESO = '\b', TAB = '\t',
               NVALIN = '\n', VTAB = '\v', RETURN = '\r'};

enum months { ENE = 1, FEB, MAR, ABR, MAY, JUN,
              JUL, AGO, SEP, OCT, NOV, DIC};
              /* FEB es 2, MAR es 3, etc. */
```

Los nombres que están en enumeraciones diferentes deben ser distintos. Los valores no necesitan ser distintos dentro de la misma enumeración.

Las enumeraciones proporcionan una manera conveniente de asociar valores constantes con nombres, una alternativa a #define con la ventaja de que los valores pueden ser generados para uno. Aunque las variables de tipos enum pueden ser declaradas, los compiladores no necesitan revisar que lo que se va a almacenar en tal variable es un valor válido para la enumeración. No obstante, las variables de enumeración ofrecen la oportunidad de revisarlas y tal cosa es a menudo mejor que #define. Además, un depurador puede ser capaz de imprimir los valores de variables de enumeración en su forma simbólica.

2.4 Declaraciones

Todas las variables deben ser declaradas antes de su uso, aunque ciertas declaraciones pueden ser hechas en forma implícita por el contexto. Una declaración especifica un tipo, y contiene una lista de una o más variables de ese tipo, como en

```
int lower, upper, step;
char c, line [1000];
```

Las variables pueden ser distribuidas entre las declaraciones en cualquier forma; la lista de arriba podría igualmente ser escrita como

```
int  lower;
int  upper;
int  step;
char c;
char line[1000];
```

Esta última forma ocupa más espacio, pero es conveniente para agregar un comentario a cada declaración o para modificaciones subsecuentes.

Una variable también puede ser inicializada en su declaración. Si el nombre es seguido por un signo de igual y una expresión, la expresión sirve como un inicializador, como en


```
char esc = '\\';
int   i = 0;
int   limit = MAXLINE+1;
float eps = 1.0e-5;
```

Si la variable en cuestión no es automática, la inicialización es efectuada sólo una vez, conceptualmente antes de que el programa inicie su ejecución, y el inicializador debe ser una expresión constante. Una variable automática explícitamente inicializada es inicializada cada vez que se entra a la función o bloque en que se encuentra; el inicializador puede ser cualquier expresión. Las variables estáticas y externas son inicializadas en cero por omisión. Las variables automáticas para las que no hay un inicializador explícito tienen valores indefinidos (esto es, basura).

El calificador const puede aplicarse a la declaración de cualquier variable para especificar que su valor no será cambiado. Para un arreglo, el calificador const indica que los elementos no serán alterados.

```
const double e = 2.71828182845905;
const char msg[] = "precaución: ";
```

La declaración const también se puede utilizar con argumentos de tipo arreglo, para indicar que la función no cambia ese arreglo:

```
int strlen(const char[]);
```

Si se efectúa un intento de cambiar un const, el resultado está definido por la implantación.

2.5 Operadores aritméticos

Los operadores aritméticos binarios son +, −, *, /, y el operador módulo %. La división entera trunca cualquier parte fraccionaria. La expresión

```
x % y
```

produce el residuo cuando x es dividido entre y, por lo que es cero cuando y divide a x exactamente. Por ejemplo, un año es bisiesto si es divisible entre 4 pero no entre 100, excepto aquellos años que son divisibles entre 400, que *sí son* bisiestos. Por lo tanto

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d es un año bisiesto\n", year);
else
    printf("%d no es un año bisiesto\n", year);
```

El operador % no puede aplicarse a operandos float o double. La dirección de truncamiento para / y el signo del resultado de % son dependientes de la máquina para operandos negativos, así como la acción que se toma en caso de sobreflujo o subflujo.

Los operadores binarios + y − tienen la misma precedencia, la cual es menor que la precedencia de *, /, y %, que a su vez es menor que + y − unarios. Los operadores aritméticos se asocian de izquierda a derecha.

La tabla 2-1 que se encuentra al final de este capítulo, resume la precedencia y asociatividad para todos los operadores.

2.6 Operadores de relación y lógicos

Los operadores de relación son

```
>  >=  <  <=
```

Todos ellos tienen la misma precedencia. Precisamente bajo ellos en precedencia están los operadores de igualdad:

```
==  !=
```

Los operadores de relación tienen precedencia inferior que los operadores aritméticos, así que una expresión como `i < lim-1` se toma como `i < (lim-1)`, como se esperaría.

Más interesantes son los operadores lógicos && y ||. Las expresiones conectadas por && o || son evaluadas de izquierda a derecha, y la evaluación se detiene tan pronto como se conoce el resultado verdadero o falso. La mayoría de los programas en C descansan sobre esas propiedades. Por ejemplo, aquí está un ciclo de la función de entrada getline que escribimos en el capítulo 1:

```
for (i=0; i<lim-1 && (c=getchar( )) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Antes de leer un nuevo carácter es necesario verificar que hay espacio para almacenarlo en el arreglo s, así que la prueba `i < lim-1` *debe* hacerse primero. Además, si esta prueba falla, no debemos seguir y leer otro carácter.

De manera semejante, sería desafortunado si c fuese probada contra EOF antes de que se llame a getchar; por lo tanto, la llamada y la asignación deben ocurrir antes de que se pruebe el carácter c.

La precedencia de && es más alta que la de ||, y ambas son menores que los operadores de relación y de asignación, así que expresiones como

```
i<lim-1 && (c = getchar( )) != '\n' && c != EOF
```

no requieren de paréntesis adicionales. Pero puesto que la precedencia de != es superior que la asignación, los paréntesis se necesitan en

```
(c = getchar( )) != '\n'
```

para obtener el resultado deseado de asignación a c y después comparación con '\n'.

Por definición, el valor numérico de una expresión de relación o lógica es 1 si la relación es verdadera, y 0 si la relación es falsa.

El operador unario de negación ! convierte a un operando que no es cero en 0, y a un operando cero en 1. Un uso común de ! es en construcciones como

```
if (!válido)
```

en lugar de

```
if (válido == 0)
```

Es difícil generalizar acerca de cuál es la mejor. Construcciones como !válido se leen en forma agradable ("si no es válido"), pero otras más complicadas pueden ser difíciles de entender.

Ejercicio 2-2. Escriba un ciclo equivalente a la iteración for anterior sin usar && o ||. □

2.7 Conversiones de tipo

Cuando un operador tiene operandos de tipos diferentes, éstos se convierten a un tipo común de acuerdo con un reducido número de reglas. En general, las únicas conversiones automáticas son aquellas que convierten un operando "angosto" en uno "amplio" sin pérdida de información, tal como convertir un entero a punto flotante en una expresión como `f + i`. Las expresiones que no tienen sentido, como utilizar un float como subíndice, no son permitidas. Las expresiones que podrían perder información, como asignar un tipo mayor a uno más corto, o un tipo de punto flotante a un entero, pueden producir una advertencia, pero no son ilegales.

Un char sólo es un entero pequeño, por lo que los char se pueden utilizar libremente en expresiones aritméticas. Esto permite una flexibilidad considerable en ciertas clases de transformación de caracteres. Una es ejemplificada con esta ingenua implantación de la función atoi, que convierte una cadena de dígitos en su equivalente numérico.

```
/* atoi: convierte s en entero */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Tal como se discutió en el capítulo 1, la expresión

```
s[i] - '0'
```

da el valor numérico del carácter almacenado en s[i], debido a que los valores de '0', '1', etc., forman una secuencia ascendente contigua.

Otro ejemplo de conversión de **char** a **int** es la función lower, que convierte un carácter sencillo a minúscula *para el conjunto de caracteres ASCII*. Si el carácter no es una letra mayúscula, lower lo regresa sin cambio.

```
/* lower:  convierte c a minúscula; solamente ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Esto funciona para ASCII debido a que las correspondientes letras mayúsculas y minúsculas están a una distancia fija como valores numéricos y cada alfabeto es contiguo —no hay sino letras entre **A** y **Z**. Sin embargo, esta última observación no es cierta para el conjunto de caracteres EBCDIC, así que este código podría convertir algo más que sólo letras en EBCDIC.

El *header* estándar **<ctype.h>**, que se describe en el apéndice B, define una familia de funciones que proporcionan pruebas y conversiones independientes de los juegos de caracteres. Por ejemplo, la función **tolower(c)** regresa el valor de la letra minúscula de **c** si **c** es una mayúscula, de modo que **tolower** es un reemplazo transportable para la función **lower** mostrada antes. De modo semejante, la prueba.

```
c >= '0' && c <= '9'
```

puede reemplazarse por

```
isdigit(c)
```

Nosotros utilizaremos las funciones de <ctype.h> en adelante.

Existe un sutil punto acerca de la conversión de caracteres a enteros. El lenguaje no especifica si las variables de tipo **char** son valores con o sin signo. Cuando un **char** se convierte a **int**, ¿puede producir alguna vez un entero negativo? La respuesta varía de una máquina a otra, reflejando diferencias en la arquitectura. En algunas máquinas un **char** cuyo bit más a la izquierda es 1 se convertirá a un entero negativo ("extensión de signo"). En otras, un **char** es promovido a un int agregando ceros del lado izquierdo, así que siempre es positivo.

La definición de C garantiza que ningún carácter que esté en el conjunto estándar de caracteres de impresión de la máquina será negativo, de modo que esos caracteres siempre serán cantidades positivas en las expresiones. Pero hay patrones arbitrarios de bits almacenados en variables de tipo carácter que pueden

aparecer como negativos en algunas máquinas, aunque sean positivos en otras. Por transportabilidad, se debe especificar **signed** o **unsigned** si se van a almacenar datos que no son caracteres en variables **tipo char**.

Las expresiones de relación como i > j y las expresiones lógicas conectadas por && y || están definidas para tener un valor de 1 siendo verdaderas, y 0 al ser falsas. De este modo, la asignación

```
d = c >= '0' && c <= '9'
```

hace 1 a **d** si **c** es un dígito, y 0 si no lo es. Sin embargo, las funciones como isdigit pueden regresar cualquier valor diferente de cero como verdadero. En la parte de validación de **if**, **while**, **for**, etc., "verdadero" es sólo "diferente de cero", por lo que esto no hace diferencia.

Las conversiones aritméticas implícitas trabajan como se espera. En general, si un operador como + o * que toma dos operandos (operador binario) tiene operandos de diferentes tipos, el tipo "menor" es *promovido* al tipo "superior" antes de que la operación proceda. El resultado es el del tipo mayor. La sección 6 del apéndice A establece las reglas de conversión en forma precisa. Si no hay operandos **unsigned**, sin embargo, el siguiente conjunto informal de reglas bastará:

> Si cualquier operando es **long double**, conviértase el otro a **long double**.
>
> De otra manera, si cualquier operando es **double**, conviértase el otro a **double**.
>
> De otra manera, si cualquier operando es **float**, conviértase el otro a **float**.
>
> De otra manera, conviértase **char** y **short** a **int**.
>
> Después, si cualquier operando es **long**, conviértase el otro a **long**.

Nótese que los **float** que están en una expresión no se convierten automáticamente a **double**; esto es un cambio de la definición original. En general, las funciones matemáticas como las de **<math.h>** utilizarán doble precisión. La razón principal para usar **float** es ahorrar espacio de almacenamiento en arreglos grandes o, con menor frecuencia, ahorrar tiempo en máquinas en donde la aritmética de doble precisión es particularmente costosa.

Las reglas de conversión son más complicadas cuando hay operandos **unsigned**. El problema es que las comparaciones de valores con signo y sin signo son dependientes de la máquina, debido a que dependen de los tamaños de los varios tipos de enteros. Por ejemplo, supóngase que int es de 16 bits y long de 32. Entonces -1L < 1U, debido a que 1U, que es un int, es promovido a signed long. Pero -1L > 1UL, debido a que -1L es promovido a unsigned long y así parece ser un gran número positivo.

Las conversiones también tienen lugar en las asignaciones; el valor del lado derecho es convertido al tipo de la izquierda, el cual es el tipo del resultado.

Un carácter es convertido a un entero, tenga o no extensión de signo, como se describió anteriormente.

Los enteros más largos son convertidos a cortos o a **char** desechando el exceso de bits de más alto orden. Así en

```
int  i;
char c;

i = c;
c = i;
```

el valor de **c** no cambia. Esto es verdadero ya sea que se inmiscuya o no la extensión de signo. Sin embargo, el invertir el orden de las asignaciones podría producir pérdida de información.

Si **x** es **float** e **i** es **int**, entonces **x = i** e **i = x** producirán conversiones; de **float** a **int** provoca el truncamiento de cualquier parte fraccionaria. Cuando **double** se convierte a **float**, el que se redondee o trunque el valor es dependiente de la implantación.

Puesto que un argumento de la llamada a una función es una expresión, también suceden conversiones de tipo cuando se pasan argumentos a funciones. En ausencia del prototipo de una función, **char** y **short** pasan a ser **int**, y **float** se hace doble. Esta es la razón por la que hemos declarado los argumentos a funciones como **int** y **double**, aun cuando la función se llama con **char** y **float**.

Finalmente, la conversión explícita de tipo puede ser forzada ("coaccionada") en cualquier expresión, con un operador unario llamado *cast*. En la construcción

(*nombre-de-tipo*) *expresión*

la *expresión* es convertida al tipo nombrado, por las reglas de conversión anteriores. El significadoprecisode un *cast* es como si la *expresión* fuera asignada a una variable del tipo especificado, que se utiliza entonces en lugar de la construcción completa. Por ejemplo, la rutina de biblioteca **sqrt** espera un argumento de doble precisión y producirá resultados sin sentido si maneja inadvertidamente algo diferente. (**sqrt** está declarado en <math.h>.) Así, si **n** es un entero, podemos usar

```
sqrt((double) n)
```

para convertir el valor de **n** a doble antes de pasarlo a **sqrt**. Nótese que la conversión forzosa produce el *valor* de **n** en el tipo apropiado; **n** en sí no se altera. El operador *cast* tiene la misma alta precedencia que otros operadores unarios, como se resume en la tabla del final de este capítulo.

Si un prototipo de función declara argumentos, como debe ser normalmente, la declaración produce conversión forzada automática de los argumentos cuando la función es llamada. Así, dado el prototipo de la función **sqrt**:

```
double sqrt(double);
```

la llamada

```
raiz2 = sqrt(2);
```

obliga al entero **2** a ser el valor **double 2.0** sin necesidad de ningún *cast*.

La biblioteca estándar incluye una implantación transportable de un generador de números pseudoaleatorios, y una función para inicializar la semilla; lo primero ilustra un *cast*:

```
unsigned long int next = 1;

/* rand:  regresa un entero pseudoaleatorio en 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand:  fija la semilla para rand( ) */
void srand(unsigned int seed)
{
    next = seed;
}
```

Ejercicio 2-3. Escriba la función **htoi(s)**, que convierte una cadena de dígitos hexadecimales (incluyendo **0x** ó **0X** en forma optativa) en su valor entero equivalente. Los dígitos permitidos son del **0** al **9**, de la **a** a la **f**, y de la **A** a la **F**. □

2.8 Operadores de incremento y decremento

El lenguaje C proporciona dos operadores poco comunes para incrementar y decrementar variables. El operador de aumento **++** agrega **1** a su operando, en tanto que el operador de disminución **--** le resta 1. Hemos usado frecuentemente **++** para incrementar variables, como en

```
if (c == '\n')
    ++nl;
```

El aspecto poco común es que **++** y **--** pueden ser utilizado como prefijos (antes de la variable, como en **++n**), o como postfijos (después de la variable: **n++**). En ambos casos, el efecto es incrementar **n**. Pero la expresión **++n** incrementa a **n** *antes* de que su valor se utilice, en tanto que **n++** incrementa a

n *después* de que su valor se ha empleado. Esto significa que en un contexto donde el valor está siendo utilizado, y no sólo el efecto, ++n y n++ son diferentes. Si n es 5, entonces

```
x = n++;
```

asigna 5 a x, pero

```
x = ++n;
```

hace que **x** sea 6. En ambos casos, n se hace 6. Los operadores de incremento y decremento sólo pueden aplicarse a variables; una expresión como (i+j)++ es ilegal.

Dentro de un contexto en donde no se desea ningún valor, sino sólo el efecto de incremento, como en

```
if (c == '\n')
    nl++;
```

prefijos y postfijos son iguales. Pero existen situaciones en donde se requiere específicamente uno u otro. Por ejemplo, considérese la función **squeeze(s,c)**, que elimina todas las ocurrencias del carácter c de una cadena s.

```
/* squeeze:  borra todas las c de s */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Cada vez que se encuentra un valor diferente de **c**, éste se copia en la posición actual j, y sólo entonces j es incrementada para prepararla para el siguiente carácter. Esto es exactamente equivalente a

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Otro ejemplo de construcción semejante viene de la función **getline** que escribimos en el capítulo 1, en donde podemos reemplazar

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```


por algo más compacto como

```
if (c == '\n')
    s[i++] = c;
```

Como un tercer ejemplo, considérese que la función estándar strcat(s, t), que concatena la cadena t al final de la cadena s. **strcat** supone que hay suficiente espacio en s para almacenar la combinación. Como la hemos escrito, **strcat** no regresa un valor; la versión de la biblioteca estándar regresa un apuntador a la cadena resultante.

```
/* strcat: concatena t al final de s; s debe ser suficientemente grande */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0')    /* encuentra el fin de s */
        i++;
    while ((s[i++] = t[j++]) != '\0')    /* copia t */
        ;
}
```

Como cada carácter se copia de t a s, el ++ postfijo se aplica tanto a i como a j para estar seguros de que ambos están en posición para la siguiente iteración.

Ejercicio 2-4. Escriba una versión alterna de **squeeze(s1,s2)** que borre cada carácter de s1 que coincida con cualquier carácter de la *cadena* s2. □

Ejercicio 2-5. Escriba la función **any(s1,s2)**, que regresa la primera posición de la cadena s1 en donde se encuentre cualquier carácter de la cadena s2, o −1 si s1 no contiene caracteres de s2. (La función de biblioteca estándar **strpbrk** hace el mismo trabajo pero regresa un apuntador a la posición encontrada.) □

2.9 Operadores para manejo de bits

El lenguaje C proporciona seis operadores para manejo de bits; sólo pueden ser aplicados a operandos integrales, esto es, char, short, int, y long, con o sin signo.

&	AND de bits
|	OR inclusivo de bits
^	OR exclusivo de bits
<<	corrimiento a la izquierda
>>	corrimiento a la derecha
~	complemento a uno (unario)

El operador AND de bits & a menudo es usado para enmascarar algún conjunto de bits; por ejemplo,

```
n = n & 0177;
```

hace cero todos los bits de n, menos los 7 de menor orden.

El operador OR de bits | es empleado para encender bits:

```
x = x | SET_ON;
```

fija en uno a todos los bits de x que son uno en SET_ON.

El operador OR exclusivo ^ pone un uno en cada posición en donde sus operandos tienen bits diferentes, y cero en donde son iguales.

Se deben distinguir los operadores de bits & y | de los operadores lógicos && y ||, que implican evaluación de izquierda a derecha de un valor de verdad. Por ejemplo, si x es 1 y y es 2, entonces x & y es cero en tanto que x && y es uno.

Los operadores de corrimiento << y >> realizan corrimientos a la izquierda y a la derecha de su operando que está a la izquierda, el número de posiciones de bits dado por el operando de la derecha, el cual debe ser positivo. Así x << 2 desplaza el valor de x a la izquierda dos posiciones, llenando los bits vacantes con cero; esto es equivalente a una multiplicación por 4. El correr a la derecha una cantidad *unsigned* siempre llena los bits vacantes con cero. El correr a la derecha una cantidad signada llenará con bits de signo ("corrimiento aritmético") en algunas máquinas y con bits 0 ("corrimiento lógico") en otras.

El operador unario ~ da el complemento a uno de un entero; esto es, convierte cada bit 1 en un bit 0 y viceversa. Por ejemplo,

```
x = x & ~077
```

fija los últimos seis bits de x en cero. Nótese que x & ~077 es independiente de la longitud de la palabra, y por lo tanto, es preferible a, por ejemplo, x & 0177700, que supone que x es una cantidad de 16 bits. La forma transportable no involucra un costo extra, puesto que ~077 es una expresión constante que puede ser evaluada en tiempo de compilación.

Como ilustración de algunos de los operadores de bits, considere la función getbits(x,p,n) que regresa el campo de n bits de x (ajustado a la derecha) que principia en la posición p. Se supone que la posición del bit 0 está en el borde derecho y que n y p son valores positivos adecuados. Por ejemplo, getbits(x,4,3) regresa los tres bits que están en la posición 4, 3 y 2, ajustados a la derecha.

```
/* getbits:  obtiene n bits desde la posición p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

La expresión x >> (p+1−n) mueve el campo deseado al borde derecho de la palabra. ~0 es todos los bits en 1; corriendo n bits hacia la izquierda con ~0<<n coloca ceros en los n bits más a la derecha; complementado con ~ hace una máscara de unos en los n bits más a la derecha.

Ejercicio 2-6. Escriba una función setbits(x,p,n,y) que regresa x con los n bits que principian en la posición p iguales a los n bits más a la derecha de y, dejando los otros bits sin cambio. □

Ejercicio 2-7. Escriba una función invert(x,p,n) que regresa x con los n bits que principian en la posición p invertidos (esto es, 1 cambiado a 0 y viceversa), dejando los otros sin cambio. □

Ejercicio 2-8. Escriba una función rightrot(x,n) que regresa el valor del entero x rotado a la derecha n posiciones de bits. □

2.10 Operadores de asignación y expresiones

Las expresiones tales como

```
i = i + 2
```

en las que la variable del lado izquierdo se repite inmediatamente en el derecho, pueden ser escritas en la forma compacta

```
i += 2
```

El operador += se llama *operador de asignación.*

La mayoría de los operadores binarios (operadores como + que tienen un operando izquierdo y otro derecho) tienen un correspondiente operador de asignación op=, en donde op es uno de

```
+ - * / % << >> & ^ |
```

Si $expr_1$ y $expr_2$ son expresiones, entonces

$expr_1$ *op* = $expr_2$

es equivalente a

$expr_1$ = ($expr_1$) *op* ($expr_2$)

exceptuando que $expr_1$ se calcula sólo una vez. Nótense los paréntesis alrededor de $expr_2$:

```
x *= y + 1
```

significa

```
x = x * (y + 1)
```

y no

```
x = x * y + 1
```

Como ejemplo, la función *bitcount* cuenta el número de bits en 1 en su argumento entero.

```
/* bitcount:  cuenta bits 1 en x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Declarar al argumento **x** como **unsigned** asegura que cuando se corre a la derecha, los bits vacantes se llenarán con ceros, no con bits de signo, sin importar la máquina en la que se ejecute el programa.

Muy aparte de su concisión, los operadores de asignación tienen la ventaja de que corresponden mejor con la forma en que la gente piensa. Decimos "suma 2 a i" o "incrementa i en 2", no "toma i, agrégale 2, después pon el resultado de nuevo en i". Así la expresión i += 2 es preferible a i = i + 2. Además, para una expresión complicada como

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

el operador de asignación hace al código más fácil de entender, puesto que el lector no tiene que verificar arduamente que dos expresiones muy largas son en realidad iguales, o preguntarse por qué no lo son, y un operador de asignación puede incluso ayudar al compilador a producir código más eficiente.

Ya hemos visto que la proposición de asignación tiene un valor y puede estar dentro de expresiones; el ejemplo más común es

```
while ((c = getchar( )) != EOF)
    ...
```

Los otros operadores de asignación (+=, -=, etc.) también pueden estar dentro de expresiones, aunque esto es menos frecuente.

En todas esas expresiones, el tipo de una expresión de asignación es el tipo de su operando del lado izquierdo, y su valor es el valor después de la asignación.

Ejercicio 2-9. En un sistema de números de complemento a dos, **x &= (x−1)** borra el bit 1 de más a la derecha en **x**. Explique el porqué. Utilice esta observación para escribir una versión más rápida de **bitcount**. □

2.11 Expresiones condicionales

Las proposiciones

```
if (a > b)
    z = a;
else
    z = b;
```


calculan en **z** el máximo de **a** y **b**. La *expresión condicional*, escrita con el operador ternario "?:", proporciona una forma alternativa para escribir ésta y otras construcciones semejantes. En la expresión

$expr_1$? $expr_2$: $expr_3$

la expresión $expr_1$ es evaluada primero. Si es diferente de cero (verdadero), entonces la expresión $expr_2$ es evaluada, y ése es el valor de la expresión condicional. De otra forma, $expr_3$ se evalúa, y ése es el valor. Sólo uno de entre $expr_2$ y $expr_3$, se evalúa. Así, para hacer **z** el máximo de **a** y **b**,

```
z = (a > b) ? a : b;      /* z = máx(a,b) */
```

Se debe notar que la expresión condicional es en sí una expresión, y se puede utilizar en cualquier lugar donde otra expresión pueda. Si $expr_2$ y $expr_3$, son de tipos diferentes, el tipo del resultado se determina por las reglas de conversión discutidas anteriormente en este capítulo. Por ejemplo, si **f** es un **float** y **n** es un **int**, entonces la expresión

```
(n > 0) ? f : n
```

es de tipo **float** sea **n** positivo o no.

Los paréntesis no son necesarios alrededor de la primera expresión de una expresión condicional, puesto que la precedencia de ?: es muy baja, sólo arriba de la asignación. De cualquier modo son recomendables, puesto que hacen más fácil de ver la parte de condición de la expresión.

La expresión condicional frecuentemente lleva a un código conciso. Por ejemplo, este ciclo imprime **n** elementos de un arreglo, 10 por línea, con cada columna separada por un blanco, y con cada línea (incluida la última) terminada por una nueva línea.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

Se imprime un carácter nueva línea después de cada diez elementos, y después del n-ésimo. Todos los otros elementos son seguidos por un espacio en blanco. Esto podría verse oscuro, pero es más compacto que el **if-else** equivalente. Otro buen ejemplo es

```
printf("Hay %d elemento%s.\n", n, n==1 ? "" : "s");
```

Ejercicio 2-10. Reescriba la función **lower**, que convierte letras mayúsculas e minúsculas, con una expresión condicional en vez de un if-else. □

2.12 Precedencia y orden de evaluación

La tabla 2-1 resume las reglas de precedencia y asociatividad de todos los operadores, incluyendo aquellos que aún no se han tratado. Los operadores que están en la misma línea tienen la misma precedencia; los renglones están en orden

de precedencia decreciente, así, por ejemplo, *,/, y % tienen todos la misma precedencia, la cual es más alta que la de + y − binarios. El "operador" ( ) se refiere a la llamada a una función. Los operadores −> y . son utilizados para tener acceso a miembros de estructuras; serán cubiertos en el capítulo 6, junto con **sizeof** (tamaño de un objeto). En el capítulo 5 se discuten * (indirección a través de un apuntador) y & (dirección de un objeto), y en el capítulo 3 se trata al operador coma.

Nótese que la precedencia de los operadores de bits &, ^, y | está debajo de = = y != . Esto implica que las expresiones de prueba de bits como

```
if ((x & MASK) == 0) ...
```

deben ser completamente colocadas entre paréntesis para dar los resultados apropiados.

TABLA 2-1. PRECEDENCIA Y ASOCIATIVIDAD DE OPERADORES

OPERADORES	ASOCIATIVIDAD
( ) [ ] ->	izquierda a derecha
! ~ ++ -- + - * & (*tipo*) sizeof	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= /= %= &= ^= |= <<= >>=	derecha a izquierda
,	izquierda a derecha

Los +, −, y * unarios, tienen mayor precedencia que las formas binarias.

Como muchos lenguajes, C no especifica el orden en el cual los operandos de un operador serán evaluados. (Las excepciones son &&, ||, ?: y ','.) Por ejemplo, en proposiciones como

```
x = f( ) + g( );
```

f puede ser evaluada antes de **g** o viceversa; de este modo si **f** o **g** alteran una variable de la que la otra depende, **x** puede depender del orden de evaluación. Se pueden almacenar resultados intermedios en variables temporales para asegurar una secuencia particular.

De manera semejante, el orden en el que se evalúan los argumentos de una función no está especificado, de modo que la proposición

```
printf("%d %d\n", ++n, power(2, n));     /* EQUIVOCADO */
```

puede producir resultados diferentes con distintos compiladores, dependiendo de si **n** es incrementada antes de que se llame a **power**. La solución, por supuesto, es escribir

```
++n;
printf("%d %d\n", n, power(2, n));
```

Las llamadas a funciones, proposiciones de asignación anidadas, y los operadores de incremento y decremento provocan "efectos colaterales" —alguna variable es modificada como producto de la evaluación de una expresión. En cualquier expresión que involucra efectos colaterales, pueden existir sutiles dependencias del orden en que las variables involucradas en la expresión se actualizan. La infortunada situación es tipificada por la proposición

```
a[i] = i++;
```

La pregunta es si el subíndice es el viejo o el nuevo valor de **i**. Los compiladores pueden interpretar esto en formas diferentes, y generar diferentes respuestas dependiendo de su interpretación. El estándar deja intencionalmente sin especificación la mayoría de tales aspectos. Cuando hay efectos colaterales (asignación a variables) dentro de una expresión, se deja a la prudencia del compilador, puesto que el mejor orden depende grandemente de la arquitectura de la máquina. (El estándar sí especifica que todos los efectos colaterales sobre argumentos sucedan antes de que la función sea llamada, pero eso podría no ayudar en la llamada a **printf** mostrada anteriormente.)

La moraleja es que escribir un código que dependa del orden de evaluación es una mala práctica de programación en cualquier lenguaje. Naturalmente, es necesario conocer qué cosas evitar, pero si no sabe *cómo* se hacen las cosas en varias máquinas, no debe intentar aprovechar una implantación en particular.

CAPITULO 3: Control de flujo

Las proposiciones de control de flujo de un lenguaje especifican el orden en que se realiza el procesamiento. Ya hemos visto la mayoría de las construcciones de control de flujo en ejemplos anteriores; aquí completaremos el conjunto, y seremos más precisos acerca de las discutidas con anterioridad.

3.1 Proposiciones y bloques

Una expresión como **x = 0** ó i+ + o printf(...) se convierte en una *proposición* cuando va seguida de un punto y coma, como en

```
x = 0;
i++;
printf (...);
```

En C, el punto y coma es un terminador de proposición, en lugar de un separador, como lo es en un lenguaje tipo Pascal.

Las llaves { y } se emplean para agrupar declaraciones y proposiciones dentro de una *proposición compuesta* o *bloque*, de modo que son sintácticamente equivalentes a una proposición sencilla. Las llaves que encierran las proposiciones de una función son un ejemplo obvio; otros ejemplos son las llaves alrededor de proposiciones múltiples después de un if, **else**, **while** o **for**. (Pueden declararse variables dentro de *cualquier* bloque; esto se expondrá en el capítulo 4.) No hay punto y coma después de la llave derecha que termina un bloque.

3.2 If-else

La proposición **if-else** se utiliza para expresar decisiones. Formalmente, la sintaxis es

```
if (expresión)
    proposición₁
else
    proposición₂
```

donde la parte del **else** es optativa. La *expresión* se evalúa; si es verdadera (esto es, si la *expresión* tiene un valor diferente de cero), la *proposición*$_1$ se ejecuta. Si es falsa (*expresión* es cero) y si existe una parte de **else**, la *proposición*$_2$ se ejecuta en su lugar.

Puesto que un if simplemente prueba el valor numérico de una expresión, son posibles ciertas abreviaciones de código. Lo más obvio es escribir

```
if (expresión)
```

en lugar de

```
if (expresión != 0)
```

Algunas veces esto es claro y natural; otras puede ser misterioso.

Debido a que la parte **else** de un **if-else** es optativa, existe una ambigüedad cuando un **else** se omite de una secuencia if anidada. Esto se resuelve al asociar el **else** con el if anterior sin **else** más cercano. Por ejemplo, en

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

el else va con el if más interno, como se muestra con el sangrado. Si eso no es lo que se desea, se deben utilizar llaves para forzar la asociación correcta:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

La ambigüedad es especialmente perniciosa en situaciones como esta:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else        /* MAL */
    printf("error — n es negativo \n");
```

El sangrado muestra en forma inequívoca lo que se desea, pero el compilador no entiende el mensaje y asocia el else con el if más interno. Puede ser difícil encontrar esta clase de errores; es una buena idea utilizar llaves cuando hay varios if anidados.

A propósito, nótese que hay un punto y coma después de **z = a** en

```
if (a > b)
    z = a;
else
    z = b;
```

Esto se debe a que gramaticalmente al if le sigue una *proposición*, y una expresión como "z = a;" siempre se termina con punto y coma.

3.3 Else-if

La construcción

```
if (expresión)
    proposición
else if (expresión)
    proposición
else if (expresión)
    proposición
else if (expresión)
    proposición
else
    proposición
```

ocurre de modo tan frecuente que bien vale una pequeña discusión aparte. Esta secuencia de proposiciones if es la forma más general de escribir una decisión múltiple. Las *expresiones* se evalúan en orden; si cualquier *expresión* es verdadera, la *proposición* asociada con ella se ejecuta, y esto termina toda la cadena. Como siempre, el código para cada *proposición* es una proposición simple o un grupo dentro de llaves.

La parte del último **else** maneja el caso "ninguno de los anteriores" o caso por omisión cuando ninguna de las otras condiciones se satisface. En algunos casos no hay una acción explícita para la omisión; en ese caso el

```
else
    proposición
```

del final puede omitirse, o puede utilizarse para detección de errores al atrapar una condición "imposible".

Para ilustrar una decisión de tres vías, se muestra una función de búsqueda binaria que decide si un valor particular de **x** se encuentra en el arreglo ordenado **v**. Los elementos de **v** deben estar en orden ascendente. La función regresa la posición (un número entre 0 y **n**−1) si **x** está en **v**, y −1 si no es así.

La búsqueda binaria primero compara el valor de entrada **x** con el elemento medio del arreglo **v**. Si **x** es menor que el valor del medio, la búsqueda se enfoca sobre la mitad inferior de la tabla; de otra manera lo hace en la mitad superior. En cualquier caso, el siguiente paso es comparar a **x** con el elemento medio de

la mitad seleccionada. Este proceso de dividir en dos continúa hasta que se encuentra el valor o ya no hay elementos.

```
/* binsearch: encuentra x en v[0]  <= v[1]  <= ...  <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v [mid])
            low = mid + 1;
        else    /* el elemento fue encontrado */
            return mid;
    }
    return -1;    /* no se encontró */
}
```

La decisión fundamental es si **x** es menor que, mayor que o igual al elemento medio **v[mid]** en cada paso; esto es un **else-if** natural.

Ejercicio 3-1. Nuestra búsqueda binaria realiza dos pruebas dentro del ciclo, cuando una podría ser suficiente (al precio de más pruebas en el exterior). Escriba una versión con sólo una prueba dentro del ciclo y mida la diferencia en tiempo de ejecución. □

3.4 Switch

La proposición switch es una decisión múltiple que prueba si una expresión coincide con uno de un número de valores *constantes* enteros, y traslada el control adecuadamente.

```
switch (expresión) {
    case exp-const: proposiciones
    case exp-const: proposiciones
    default: proposiciones
}
```

Cada case se etiqueta con uno o más valores constantes enteros o expresiones constantes enteras. Si un case coincide con el valor de la expresión, la ejecución comienza allí. Todas las expresiones case deben ser diferentes. El etiquetado como **default** se ejecuta si ninguno de los otros se satisface. El **default**

es optativo; si no está y ninguno de los casos coincide, no se toma acción alguna. Las cláusulas case y default pueden ocurrir en cualquier orden.

En el capítulo 1 se escribió un programa para contar las ocurrencias de cada dígito, espacio en blanco y todos los demás caracteres, usando una secuencia de if ... else if ... else. Aquí está el mismo programa con un switch:

```
#include <stdio.h>

main( )   /* cuenta dígitos, espacios blancos, y otros */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar( )) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("dígitos =");
    for (i = 0; i<10; i++)
        printf(" %d", ndigit[i]);
    printf(", espacios blancos = %d, otros = %d\n",
        nwhite, nother);
    return 0;
}
```

La proposición **break** provoca una salida inmediata del **switch**. Puesto que los case sirven sólo como etiquetas, después de que se ejecuta el código para uno, la ejecución pasa al siguiente, a menos que se tome una acción específica para terminar el **switch**. Las formas más comunes de dejar un **switch** son **break** y **return**. Una proposición **break** también se puede emplear para forzar una salida inmediata de los ciclos **while**, **for** y **do**, como se verá más adelante en este capítulo.

Pasar a través de los `case` es en parte bueno y en parte no. Por el lado positivo, esto permite conectar varios `case` a una acción simple, como con los dígitos de este ejemplo. Pero eso también implica que cada case normalmente debe terminar con un **break** para prevenir pasar al siguiente. Pasar de un `case` a otro no es una práctica muy sólida y es susceptible a la desintegración cuando se modifica el programa. Con la excepción de etiquetas múltiples para un cálculo simple, lo anterior se debe utilizar con cautela y emplear comentarios.

Como formalidad, coloque un **break** después del último `case` (en este caso el **default**) aun si es lógicamente innecesario. Algún día, cuando se agregue otro `case` al final, esta práctica de programación defensiva lo salvará.

Ejercicio 3-2. Escriba una función **escape(s,t)** que convierte caracteres como nueva línea y tabulación en secuencias de escape visibles como \n y \t mientras copia la cadena t a s. Utilice un **switch**. Escriba también una función para la dirección inversa, convirtiendo secuencias de escape en caracteres reales. □

3.5 Ciclos—while y for

Ya hemos encontrado los ciclos **while** y **for**. En

```
while (expresión)
    proposición
```

la *expresión* se evalúa. Si es diferente de cero, se ejecuta la *proposición* y se reevalúa la *expresión*. Este ciclo continúa hasta que la *expresión* se hace cero, punto en el cual se suspende la ejecución para continuar después de la *proposición*.

La proposición **for**

```
for (expr1; expr2; expr3)
    proposición
```

es equivalente a

```
expr1;
while (expr2) {
    proposición
    expr3;
}
```

excepto por el comportamiento de **continue** que se describe en la sección 3.7.

Gramaticalmente, las tres componentes de un ciclo for son expresiones. Por lo común, $expr_1$ y $expr_3$ son asignaciones o llamadas a función y $expr_2$ es una expresión de relación. Cualquiera de las tres partes se puede omitir, aunque deben permanecer los punto y coma. Si $expr_1$ o $expr_3$ se omite, sólo se desecha de la

expansión. Si la prueba $expr_2$ no está presente, se toma como permanentemente verdadera, así que

```
for (;;) {
    ...
}
```

es una iteración "infinita", que presumiblemente será interrumpida por otros medios, como un **break** o un **return**.

El usar **while** o **for** es principalmente cuestión de preferencia personal. Por ejemplo, en

```
while ((c = getchar( )) = = ' ' || c = = '\n' || c = = '\t')
    ;   /* ignora caracteres espaciadores */
```

no hay inicialización o reinicialización, por lo que el **while** es más natural.

El **for** se prefiere cuando existe una inicialización simple e incrementos, puesto que mantiene las proposiciones de control del ciclo juntas y visibles al principio del mismo. Esto es más obvio en

```
for (i = 0; i < n; i+ +)
    ...
```

que es la forma característica de procesar los primeros n elementos de un arreglo en C, lo análogo al ciclo DO de Fortran o al for de Pascal. Sin embargo, la analogía no es perfecta puesto que tanto el índice como el límite de un ciclo for en C pueden ser alterados desde dentro del ciclo, y la variable del índice i retiene su valor cuando las iteraciones terminan por cualquier razón. Debido a que las componentes del for son expresiones arbitrarias, sus ciclos no están restringidos a progresiones aritméticas. Por otra parte, considere que es un mal estilo incluir en las secciones de inicialización e incremento operaciones no relacionadas con esas actividades, que más bien se reservan para acciones de control del ciclo.

Como un ejemplo más amplio, aquí está otra versión de **atoi** para convertir una cadena a su equivalente numérico. Esta es ligeramente más general que la del capítulo 2; trata también los espacios en blanco previos al número, y los signos + o −. (El capítulo 4 muestra **atof**, que realiza la misma conversión para números de punto flotante.)

La estructura del programa refleja la forma de la entrada:

ignora espacios en blanco, si los hay
toma el signo, si lo hay
toma la parte entera y conviértela

Cada paso realiza su parte, y deja las cosas en forma clara para el siguiente. La totalidad del proceso termina con el primer carácter que no pueda ser parte de un número.

```
#include <ctype.h>

/* atoi: convierte s a entero; versión 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)    /* ignora espacio en blanco */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')    /* ignora el signo */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

La biblioteca estándar proporciona una función más elaborada, **strtol**, para la conversión de cadenas a enteros largos; véase la sección 5 del apéndice B.

Las ventajas de mantener centralizado el control del ciclo son aún más obvias cuando existen ciclos anidados. La siguiente función es una clasificación Shell para ordenar un arreglo de enteros. La idea básica de este algoritmo de ordenamiento, que fue inventado en 1959 por D.L. Shell, es que en las primeras etapas sean comparados elementos lejanos, en lugar de los adyacentes, como en los ordenamientos de intercambio más simples. Esto tiende a eliminar rápidamente gran cantidad de desorden, así que los estados posteriores tienen menos trabajo por hacer. El intervalo entre los elementos comparados disminuye en forma gradual hasta uno, punto en el que el ordenamiento viene a ser efectivamente un método adyacente de intercambio.

```
/* shellsort:  ordena v[0]...v[n-1] en orden ascendente */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
}
```

Existen tres ciclos anidados. El más externo controla el espacio entre los elementos comparados, reduciéndolo desde **n/2** por un factor de dos en cada paso hasta que llega a cero. El ciclo intermedio recorre los elementos. El ciclo más interno compara cada pareja de elementos que está separada por el espacio **gap** e invierte a las que estén desordenadas. Puesto que **gap** finalmente se reduce a uno, todos los elementos se ordenan correctamente. Nótese cómo la generalidad del **for** hace que el ciclo más externo coincida con la forma de los otros, aun cuando no es una progresión aritmética.

Un último operador de C es la coma ",", que frecuentemente encuentra uso en la proposición **for**. Una pareja de expresiones separadas por una coma se evalúa de izquierda a derecha, y el tipo y valor del resultado son el tipo y valor del operando derecho. Así, en una proposición **for** es posible colocar expresiones múltiples en las diferentes partes, por ejemplo, para procesar dos índices en paralelo. Esto se ilustra en la función **reverse(s)**, que invierte a la cadena **s** en el mismo lugar.

```
#include <string.h>

/* reverse:  invierte la cadena s en el mismo lugar */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Las comas que separan a los argumentos de una función, las variables en declaraciones, etc., *no* son operadores coma, y no garantizan evaluación de izquierda a derecha.

Los operadores coma deberán utilizarse poco. Los usos más adecuados son en construcciones fuertemente relacionadas una con la otra, como en el ciclo **for** de **reverse**, y en macros en donde un cálculo de paso múltiple debe ser una expresión simple. Una expresión coma podría también ser apropiada para el intercambio de elementos en **reverse**, donde el intercambio puede ser a través de una operación simple:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Ejercicio 3-3. Escriba la función **expand(s1,s2)** que expande notación abreviada como **a-z**, que viene en la cadena **s1**, en la lista equivalente completa **abc...xyz**, en **s2**. Permita letras mayúsculas y minúsculas, así como dígitos, y esté preparado para manejar casos como **a-b-c** y **a-z0-9** y **-a-z**. Haga que los guiones al inicio o al final se tomen literalmente. □

3.6 Ciclos—do-while

Como ya se expuso en el capítulo 1, los ciclos while y for verifican al principio la condición de término. En contraste, el tercer ciclo en C, el **do-while**, prueba al final *después* de realizar cada paso a través del cuerpo del ciclo, el cual se ejecuta siempre por lo menos una vez.

La sintaxis del do es

```
do
    proposición
while (expresión);
```

La *proposición* se ejecuta y después se evalúa la *expresión*. Si es verdadera, la *proposición* se evalúa de nuevo, y así sucesivamente. Cuando la *expresión* se hace falsa, el ciclo termina. Excepto por el sentido de la prueba, el do-while es equivalente a la proposición **repeat-until** de Pascal.

La experiencia demuestra que el **do-while** es mucho menos utilizado que el **while** y el **for**. Aunque de cuando en cuando es valioso, como en la siguiente función itoa, que convierte un número a una cadena de caracteres (lo inverso de atoi). El trabajo es ligeramente más complicado de lo que podría pensarse en un principio, debido a que los métodos fáciles para generar dígitos los generan en el orden incorrecto. Hemos elegido generar la cadena al revés y después invertirla.

```
/* itoa:  convierte n a caracteres en s */
void itoa(int n, char s[])
{
    int, i, sign;

    if ((sign = n) < 0)   /* registra el signo */
        n = -n;           /  vuelve a n positivo */
    i = 0;
    do {          /* genera dígitos en orden inverso */
        s[i++] = n % 10 + '0';   /* toma el siguiente dígito */
    } while ((n /= 10) > 0);      /* bórralo */
    if (sign < 0)
        s[i++] = '—';
    s[i] = '\0';
    reverse(s);
}
```

El do-while es necesario, o al menos conveniente, puesto que por lo menos se debe instalar un carácter en el arreglo s, aun si n es cero. También empleamos llaves alrededor de la proposición simple que hace el cuerpo del do-while, aunque son innecesarias, y así el lector apresurado no confundirá la sección del while con el *principio* de un ciclo while.

Ejercicio 3-4. En una representación de números en complemento a dos, nuestra versión de itoa no maneja el número negativo más grande, esto es, el valor de n igual a $-(2^{\text{tamaño palabra}-1})$. Explique por qué. Modifíquelo para imprimir el valor correctamente, sin importar la máquina en que ejecute. □

Ejercicio 3-5. Escriba la función itob(n,s,b) que convierte al entero n en una representación de caracteres con base **b** dentro de la cadena **s**. En particular, itob(n, s, 16) da formato a n como un entero hexadecimal en s. □

Ejercicio 3-6. Escriba una versión de itoa que acepte tres argumentos en lugar de dos. El tercer argumento es un ancho mínimo de campo; al número convertido se deben agregar blancos a la izquierda si es necesario para hacerlo suficientemente ancho. □

3.7 Break y continue

Algunas veces es conveniente tener la posibilidad de abandonar un ciclo de otra manera que no sea probando al inicio o al final. La proposición **break** proporciona una salida anticipada de un **for**, **while** y **do**, tal como lo hace el **switch**. Un break provoca que el ciclo o switch más interno que lo encierra termine inmediatamente.

La siguiente función, **trim**, elimina espacios blancos, tabuladores y nuevas líneas al final de una cadena, utilizando un **break** para salir de un ciclo cuando se encuentra el no-blanco, no-tabulador o no-nueva línea de más a la derecha.

```
/* trim:  elimina blancos, tabuladores y nueva línea al final */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

strlen regresa la longitud de la cadena. El ciclo for inicia al final y rastrea hacia atrás, buscando el primer carácter que no sea blanco o tabulador o nueva línea. El ciclo se interrumpe cuando se encuentra alguno o cuando n se hace negativa (esto es, cuando se ha rastreado toda la cadena. Se deberá verificar que este comportamiento es correcto, aun cuando la cadena esté vacía o sólo contiene espacios en blanco.

La proposición **continue** está relacionada con el **break**, pero se utiliza menos; provoca que inicie la siguiente iteración del ciclo **for**, **while** o **do** que la contiene. Dentro de **while** y **do**, esto significa que la parte de la prueba se ejecuta inmediatamente; en el **for**, el control se traslada al paso de incremento. La propo-

sición continue se aplica solamente a ciclos, no a switch. Un continue dentro de un switch que está a su vez en un ciclo, provoca la siguiente iteración del ciclo.

Como un ejemplo, el siguiente fragmento procesa sólo los elementos no negativos que están en el arreglo a; los valores negativos son ignorados.

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)    /* ignora elementos negativos */
        continue;
    ...  /* trabaja con elementos positivos */
}
```

La proposición continue se emplea a menudo cuando la parte del ciclo que sigue es complicada, de modo que invertir la prueba y sangrar otro nivel podría anidar profundamente el programa.

3.8 Goto y etiquetas

C proporciona la infinitamente abusable proposición goto, y etiquetas para saltar hacia ellas. Formalmente, el goto nunca es necesario, y en la práctica es casi siempre más fácil escribir código sin él. En este libro no se ha usado goto alguno.

Sin embargo, hay algunas situaciones donde los goto pueden encontrar un lugar. La más común es abandonar el procesamiento en alguna estructura profundamente anidada, tal como salir de dos o más ciclos a la vez. La proposición break no se puede utilizar directamente, puesto que sólo sale del ciclo más interno. Así:

```
    for ( ... )
        for ( ... ) {
            ...
            if (desastre)
                goto error;
        }
    ...
error:
    arregla el desorden
```

Esta organización es útil si el código de manejo de error no es trivial y si los errores pueden ocurrir en varios lugares.

Una etiqueta tiene la misma forma que un nombre de variable y es seguida por dos puntos. Puede ser adherida a cualquier proposición de la misma función en la que está el goto. El alcance de una etiqueta es toda la función.

Como otro ejemplo, considérese el problema de determinar si dos arreglos, a y b, tienen un elemento en común. Una posibilidad es


```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto encontrado;
    /* no se encontró ningún elemento en común */
    ...
encontrado:
    /* se tiene uno:    a[i] == b[j] */
    ...
```

El código que involucra un goto siempre puede escribirse sin él, aunque tal vez al precio de algunas pruebas repetidas o variables extra. Por ejemplo, la búsqueda en los arreglos quedará

```
encontrado = 0;
for (i = 0; i < n && !encontrado; i++)
    for (j = 0; j < m && !encontrado; j++)
        if (a[i] == b[j])
            encontrado = 1;
if (encontrado)
    /* se tiene uno:    a[i-1] == b[j-1] */
    ...
else
    /* no se encontró algún elemento en común */
    ...
```

Con pocas excepciones, como las citadas aquí, el código que se basa en proposiciones goto es generalmente más difícil de entender y de mantener que el código sin ellas. Aunque no somos dogmáticos acerca del asunto, se ve que las proposiciones goto deben ser utilizadas raramente, si acaso.

CAPITULO 4:

Funciones y la estructura del programa

Las funciones dividen tareas grandes de computación en varias más pequeñas, y permiten la posibilidad de construir sobre lo que otros ya han hecho, en lugar de comenzar desde cero. Las funciones apropiadas ocultan los detalles de operación de las partes del programa que no necesitan saber acerca de ellos, así que dan claridad a la totalidad y facilitan la penosa tarea de hacer cambios.

El lenguaje C se diseñó para hacer que las funciones fueran eficientes y fáciles de usar; los programas escritos en C se componen de muchas funciones pequeñas en lugar de sólo algunas grandes. Un programa puede residir en uno o más archivos fuente, los cuales pueden compilarse por separado y cargarse junto con funciones de biblioteca previamente compiladas. No trataremos aquí tales procesos, puesto que los detalles varían de un sistema a otro.

La declaración y definición de funciones es el área donde el estándar ANSI ha hecho los cambios más visibles a C. Tal como mencionamos en el capítulo 1, ahora es posible declarar los tipos de los argumentos cuando se declara una función. La sintaxis de la definición de funciones también cambia, de modo que las declaraciones y las definiciones coincidan. Esto hace posible que el compilador pueda detectar muchos más errores de lo que podía anteriormente. Además, cuando los argumentos se declaran con propiedad, se realizan automáticamente las conversiones convenientes.

El estándar clarifica las reglas sobre el alcance de los nombres; en particular, requiere que sólo haya una definición de cada objeto externo. La inicialización es más general: los arreglos y las estructuras automáticas ahora se pueden inicializar.

El preprocesador de C también se ha mejorado. Las nuevas facilidades del procesador incluyen un conjunto más completo de directivas para la compilación condicional, una forma de crear cadenas entrecomilladas a partir de argumentos de macros y un mejor control sobre el proceso de expansión de macros.

4.1 Conceptos básicos de funciones

Para comenzar, diseñemos y escribamos un programa que imprima cada línea de su entrada que contenga un "patrón" o cadena de caracteres en particular.

(Este es un caso especial del programa grep de UNIX.) Por ejemplo, al buscar el patrón de letras "ould" en el conjunto de líneas

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

producirá la salida

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

El trabajo se ajusta ordenadamente en tres partes:

```
while (hay otra línea)
    if (la línea contiene el patrón)
        imprímela
```

Aunque ciertamente es posible poner el código de todo esto en **main**, una mejor forma es aprovechar la estructura haciendo de cada parte una función separada. Es más fácil trabajar con tres piezas pequeñas que con una grande, debido a que los detalles irrelevantes se pueden ocultar dentro de las funciones, y minimizar así el riesgo de interacciones no deseadas. Los fragmentos incluso se pueden emplear en otros programas.

"Mientras hay otra línea" es **getline**, función que ya escribimos en el capítulo 1, e "imprímela" es **printf**, que alguien ya nos proporcionó. Esto significa que sólo necesitamos escribir una rutina para decidir si la línea contiene una ocurrencia del patrón.

Podemos resolver ese problema escribiendo una función **strindex(s,t)**, que regresa la posición o índice en la cadena **s** en donde comienza la cadena **t**, o −1 si **s** no contiene **t**. Debido a que los arreglos en C principian en la posición cero, los índices serán cero o positivos, y así un valor negativo como −1 es conveniente para señalar una falla. Cuando posteriormente se necesite una coincidencia de patrones más elaborada, sólo se debe reemplazar **strindex**; el resto del código puede permanecer igual. (La biblioteca estándar provee una función **strstr** que es semejante a **strindex**, excepto en que regresa un apuntador en lugar de un índice.)

Una vez definido todo este diseño, llenar los detalles del programa es simple. Aquí está en su totalidad, de modo que se puede ver cómo las piezas quedan juntas. Por ahora, el patrón que se buscará es una cadena literal, lo cual no es el mecanismo más general. Regresaremos en breve a una discusión sobre cómo inicializar arreglos de caracteres, y en el capítulo 5 mostraremos cómo hacer que el patrón de caracteres sea un parámetro fijado cuando se ejecuta el programa. También hay una versión ligeramente diferente de **getline**, que se podrá comparar con la del capítulo 1.


```
# include <stdio.h>
#define MAXLINE 1000      /* longitud máxima por línea de entrada */

int getline(char line[ ], int max);
int strindex(char source[], char searchfor[]);

char pattern [ ] = "ould";      /* patrón por buscar */

/*encuentra todas las líneas que coincidan con el patrón */
main ( )
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) > = 0) {
            printf("%s", line);
            found + +;
        }
    return found;
}

/* getline: trae línea y la pone en s, regresa su longitud */
int getline(char s[ ], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i+ +] = c;
    if (c = = '\n')
        s[i+ +] = c;
    s[i] = '\0';
    return i;
}

/* strindex: regresa el índice de t en s,-1 si no existe */
int strindex(char s[ ], char t[ ])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i+ +) {
        for (j=i, k=0; t[k]!='\0' && s[j]= =t[k]; j+ +, k+ +)
            ;
```

```
        if (k > o && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Cada definición de función tiene la forma

tipo-regresado nombre-de-función(declaraciones de argumentos)
{
    declaraciones y proposiciones
}

Varias partes pueden omitirse; una función mínima es

```
nada( ) { }
```

que no hace ni regresa nada. Una función hacer-nada, como ésta, es algunas veces útil para reservar lugar al desarrollar un programa. Si el tipo que regresa se omite, se supone int.

Un programa es sólo un conjunto de definiciones de variables y funciones. La comunicación entre funciones es por argumentos y valores regresados por las funciones, y a través de variables externas. Las funciones pueden presentarse en cualquier orden dentro del archivo fuente, y el programa fuente se puede dividir en varios archivos, mientras las funciones no se dividan.

La proposición **return** es el mecanismo para que la función que se llama regrese un valor a su invocador. Al **return** le puede seguir cualquier expresión:

return *expresión*

La *expresión* se convertirá al tipo de retorno de la función si es necesario. Con frecuencia se utilizan paréntesis para encerrar la *expresión*, pero son optativos.

La función que llama tiene la libertad de ignorar el valor regresado. Incluso, no hay necesidad de una expresión después de **return**; en tal caso, ningún valor regresa al invocador. También el control regresa, sin valor, cuando la ejecución "cae al final" de la función al alcanzar la llave derecha que cierra. No es ilegal, aunque probablemente un signo de problemas, el que una función regrese un valor desde un lugar y ninguno desde otro. En cualquier caso, si una función no regresa explícitamente un valor, su "valor" es ciertamente basura.

El programa de búsqueda del patrón regresa un estado desde **main**, el número de coincidencias encontradas. Este valor está disponible para ser empleado por el medio ambiente que llamó al programa.

El mecanismo de cómo compilar y cargar un programa en C que reside en varios archivos fuente varía de un sistema a otro. En el sistema UNIX, por ejemplo, la orden **cc** mencionada en el capítulo 1 hace el trabajo. Supóngase que las tres funciones se almacenan en tres archivos llamados **main.c**, **getline.c**, y **strindex.c**. Entonces la orden

```
cc main.c getline.c strindex.c
```

compila los tres archivos, sitúa el código objeto resultante en los archivos **main.o**, **getline.o**, y **strindex.o**, y después los carga todos dentro de un archivo ejecutable llamado **a.out**. Si existe un error, digamos en **main.c**, ese archivo puede volverse a compilar por sí mismo y el resultado cargado con los archivos objeto previos, con la orden.

```
cc main.c getline.o strindex.o
```

cc emplea la convención ".c" contra ".o" para distinguir los archivos fuente de los archivos objeto.

Ejercicio 4-1. Escriba la función **strrindex(s,t)**, que regresa la posición de la ocurrencia de más a la derecha de t en s, ó **−1** si no hay alguna. □

4.2 Funciones que regresan valores no enteros

Hasta ahora los ejemplos de funciones han regresado o ningún valor (void) o un **int**. ¿Qué pasa si una función debe regresar algo de otro tipo? Muchas funciones numéricas como **sqrt**, **sin** y **cos** regresan **double**; otras funciones especializadas regresan tipos diferentes. Para ilustrar cómo tratar con esto, escribamos y usemos la función **atof(s)**, que convierte la cadena **s** a su valor equivalente de punto flotante de doble precisión. La función **atof** es una extensión de **atoi**, de la que mostramos versiones en los capítulos 2 y 3. Maneja signo y punto decimal optativos, y presencia o ausencia de parte entera o fraccionaria. Nuestra versión *no es* una rutina de conversión de alta calidad; tomaría más espacio del que podemos dedicarle. La biblioteca estándar incluye un **atof**; el *header* **<math.h>** la declara.

Primero, **atof** por sí misma debe declarar el tipo del valor que regresa, puesto que no es **int**. El nombre del tipo precede al nombre de la función:

```
#include <ctype.h>

/* atof: convierte la cadena s a double */
double atof(char s[ ])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)    /* ignora espacios blancos */
        ;
    sign = (s[i] == '-') ? -1 : 1;
```

```
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

Segundo, e igualmente importante, la rutina que llama debe indicar que atof regresa un valor que no es int. Una forma de asegurar esto es declarar atof explícitamente en la rutina que la llama. La declaración se muestra en esta primitiva calculadora (apenas adecuada para un balance de chequera), que lee un número por línea, precedido en forma optativa por un signo, y lo acumula, imprimiendo la suma actual después de cada entrada:

```
#include <stdio.h>

#define MAXLINE 100

/* calculadora rudimentaria */
main( )
{
    double sum, atof(char [ ]);
    char line[MAXLINE];
    int getline(char line[ ], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

La declaración

```
double sum, atof(char [ ]);
```

señala que sum es una variable double, y que atof es una función que toma un argumento char[ ] y regresa un double.

La función atof se debe declarar y definir consistentemente. Si atof en sí misma y la llamada a ella en main tienen tipos inconsistentes dentro del mismo archivo fuente, el error será detectado por el compilador. Pero si (como es probable) atof fuera compilada separadamente, la falta de consistencia no se detectaría, atof

regresaría un valor double que main trataría como int, y se producirían resultados incongruentes.

A la luz de lo que hemos mencionado acerca de cómo deben coincidir las declaraciones con las definiciones, esto podría ser sorprendente. La razón de que ocurra una falta de coincidencia es que, si no existe el prototipo de una función, ésta es declarada implícitamente la primera vez que aparece en una expresión, como

```
sum += atof(line)
```

Si en una expresión se encuentra un nombre que no ha sido declarado previamente y está seguido por paréntesis izquierdo, se declara por contexto, de modo que se supone que es el nombre de una función que regresa un int, y nada se supone acerca de sus argumentos. Aún más, si la declaración de una función no incluye argumentos como en

```
double atof( );
```

también es tomada de modo que no se supone nada acerca de los argumentos de atof; se desactiva toda revisión de parámetros. Este significado especial de la lista de argumentos vacía se hace para permitir que los programas en C viejos se compilen con los nuevos compiladores. Pero es una mala táctica usar esto con programas nuevos. Si la función toma argumentos, declárelos; si no los toma, use void.

Dado atof, propiamente declarado, podemos escribir atoi (convierte una cadena a int) en términos de él:

```
/* atoi: convierte la cadena s a entero usando atof */
int atoi(char s[ ])
{
    double atof(char s[ ]);

    return (int) atof(s);
}
```

Nótese la estructura de las declaraciones y la proposición return. El valor de la expresión en

return *expresión*;

se convierte al tipo de la función antes de que se tome el return. Por lo tanto, el valor de atof, un double, se convierte automáticamente a int cuando aparece en este return, puesto que la función atoi regresa un int. Sin embargo, esta operación potencialmente descarta información, así que algunos compiladores lo previenen. El *cast* establece explícitamente lo que la operación intenta y suprime las advertencias.

Ejercicio 4-2. Extienda atof para que maneje notación científica de la forma

123.45e-6

donde un número de punto flotante puede ir seguido por **e** o **E** y opcionalmente un exponente con signo. □

4.3 Variables externas

Un programa en C consta de un conjunto de objetos externos, que son variables o funciones. El adjetivo "externo" se emplea en contraste con "interno", que describe los argumentos y las variables definidas dentro de las funciones. Las variables externas se definen fuera de cualquier función, y por lo tanto, están potencialmente disponibles para muchas funciones. Las funciones en sí mismas son siempre externas, puesto que C no permite definir funciones dentro de otras funciones. Por omisión, las variables y funciones externas tienen la propiedad de que todas las referencias a ellas por el mismo nombre, incluso desde funciones compiladas separadamente, son referencias a la misma cosa. (El estándar llama a esta propiedad *ligado externo.*) En este sentido, las variables externas son análogas a los bloques COMMON de Fortran o a las variables del bloque más externo de Pascal. Más adelante veremos cómo definir variables y funciones externas que sean visibles sólo dentro de un archivo fuente.

Debido a que las variables externas son accesibles globalmente, proporcionan una alternativa a los argumentos en funciones y a los valores de retorno para comunicar datos entre funciones. Cualquier función puede tener acceso a variables externas haciendo referencia a ellas solamente por su nombre, si éste ha sido declarado de alguna manera.

Si un gran número de variables se debe compartir entre funciones, las variables externas son más covenientes y eficientes que las largas listas de argumentos. Sin embargo, como se señaló en el capítulo 1, este razonamiento se deberá aplicar con precaución, pues puede tener un efecto negativo sobre la estructura del programa y dar lugar a programas con demasiadas conexiones de datos entre funciones.

Las variables externas son también útiles debido a su mayor alcance y tiempo de vida. Las variables automáticas son internas a una función y su existencia se inicia cuando se entra a la función y desaparecen cuando ésta se abandona. Por otro lado, las variables externas son permanentes, de modo que retienen sus valores de la invocación de una función a la siguiente. Así, si dos funciones deben compartir algunos datos, aun si ninguna llama a la otra, con frecuencia es más conveniente que los datos compartidos se mantengan en variables externas, en lugar de que sean pasados como argumentos de entrada y salida.

Examinemos más a fondo este tema con un ejemplo más amplio. El problema es escribir el programa de una calculadora que provea los operadores +, −, * y /. Por ser más fácil su implantación, la calculadora utilizará notación polaca inversa en lugar de infija. (La polaca inversa es utilizada por algunas calculadoras de bolsillo, y en lenguajes como Forth y Postscript.)

En notación polaca inversa, cada operador sigue a sus operandos; una expresión infija como

```
(1 - 2) * (4 + 5)
```

se introduce como

```
1 2 - 4 5 + *
```

Los paréntesis no son necesarios; la notación no es ambigua mientras sepamos cuántos operandos espera cada operador.

La implantación es simple. Cada operando se introduce en una pila o *stack*; cuando un operador llega, el número correcto de operandos (dos para operadores binarios) son extraídos, se aplica el operador y el resultado se regresa a la pila. En el ejemplo anterior, se introducen 1 y 2, después se reemplazan por su diferencia, −1. En seguida se introducen 4 y 5 y luego se reemplazan por su suma, 9. El producto de −1 y 9, que es −9, los reemplaza en la pila. El valor que se encuentra en el tope de la pila se extrae e imprime cuando se encuentra el fin de la línea de entrada.

La estructura del programa es así un ciclo que realiza las operaciones adecuadas sobre cada operador y operando que aparece:

```
while (siguiente operador u operando no es fin de archivo)
    if (número)
        introducirlo
    else if (operador)
        extraer operandos
        hacer operaciones
        introducir el resultado
    else if (nueva línea)
        extrae e imprime el tope de la pila
    else
        error
```

Las operaciones de introducir (*push*) y extraer de una pila (*pop*) son sencillas, pero cuando se les agrega detección y recuperación de errores, son suficientemente largas como para que sea mejor ponerlas en funciones separadas en lugar del código a lo largo de todo el programa. Además, debe existir una función separada para buscar el siguiente operador u operando.

La principal decisión de diseño que aún no se ha explicado es dónde está la pila, esto es, cuáles rutinas tienen acceso a ella directamente. Una posibilidad es mantenerla en **main**, y pasar la pila y la posición actual a las rutinas que introducen y extraen elementos. Pero **main** no necesita saber acerca de las variables que controlan a la pila; sólo efectúa operaciones de introducir y extraer. Así, hemos decidido almacenar la pila y su información asociada en variables externas accesibles a las funciones **push** y **pop**, pero no a **main**.

Traducir este bosquejo a código es suficientemente fácil. Si por ahora pensamos que el programa existe en un archivo fuente, se verá así:

```
#includes
#defines

declaración de funciones para main

main( ) { ... }

variables externas para push y pop

void push(double f) { ... }
double pop(void) { ... }

int getop(char s[ ]) { ... }

rutinas llamadas por getop
```

Más adelante se verá cómo esto se puede dividir entre dos o más archivos fuente.

La función **main** es un ciclo que contiene un gran **switch** sobre el tipo de operador y operando; éste es un uso del **switch** más típico que el mostrado en la sección 3.4.

```
#include <stdio.h>
#include <math.h>          /* para atof( ) */

#define MAXOP  100         /* máx tamaño de operando u operador */
#define NUMBER '0'         /* señal de que un número se encontró */

int getop(char [ ]);
void push(double);
double pop(void);

/* calculadora polaca inversa */
main( )
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop( ) + pop( ));
            break;
```



```
        case '*':
            push(pop( ) * pop( ));
            break;
        case '-':
            op2 = pop( );
            push(pop( ) - op2);
            break;
        case '/':
            op2 = pop( );
            if (op2 != 0.0)
                push(pop( ) / op2);
            else
                printf("error: divisor cero\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop( ));
            break;
        default:
            printf("error: comando desconocido %s\n", s);
            break;
        }
    }
    return 0;
}
```

Puesto que **+** y ***** son operadores conmutativos, el orden en el que se combinan los operandos extraídos es irrelevante, pero para **−** y **/** deben distinguirse los operandos izquierdo y derecho. En

```
push(pop( ) - pop( ));      /* INCORRECTO */
```

no se define el orden en el que se evalúan las dos llamadas de **pop**. Para garantizar el orden correcto, es necesario extraer el primer valor en una variable temporal, como se hizo en **main**.

```
#define MAXVAL 100          /* máxima profundidad de la pila val */

int sp = 0;                 /* siguiente posición libre en la pila */
double val[MAXVAL];         /* valores de la pila */

/* push:  introduce f a la pila */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
```

```
        else
            printf("error: pila llena, no puede efectuar push %g/n", f);
}

/* pop: extrae y regresa el valor superior de la pila */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: pila vacía\n");
        return 0.0;
    }
}
```

Una variable es externa si se encuentra definida fuera de cualquier función. Así, la pila y el índice de la pila que deben ser compartidos por **push** y por **pop** se definen fuera de estas funciones. Pero **main** en sí misma no hace referencia a la pila o a la posición de la pila —la representación puede estar oculta.

Pasemos ahora a la implantación de **getop**, la función que toma el siguiente operador u operando. La tarea es fácil. Ignorar blancos y tabuladores. Si el siguiente carácter no es un dígito o punto decimal, regresarlo. De otra manera, reunir una cadena de dígitos (que pueda incluir un punto decimal), y regresar **NUMBER**, la señal de que ha sido reunido un número.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: obtiene el siguiente operador u operando numérico */
int getop(char s[ ])
{
    int i, c;

    while ((s[0] = c = getch( )) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;          /* no es un número */
    i = 0;
    if (isdigit(c))        /* reúne la parte entera */
        while (isdigit(s[++i] = c = getch( )))
        ;
    if (c == '.') /* reúne la parte fraccionaria */
      while (isdigit(s[++i] = c = getch( )))
        ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```


¿Qué son **getch** y **ungetch**? Por lo común se da el caso de que un programa no puede determinar si ha leído suficiente de la entrada hasta que ha leído demasiado. Un ejemplo es reunir los caracteres que forman un número: hasta que se vea el primer no-dígito, el número no está completo. Pero entonces el programa ha leído un carácter de más, para el cual no está preparado.

El problema podría ser resuelto si fuera posible "desleer" el carácter no deseado. Entonces, cada vez que el programa lea un carácter de más, podría regresarlo a la entrada, así que el resto del código se podrá comportar como si nunca se hubiese leído. Afortunadamente, es fácil simular el regreso de un carácter, escribiendo un par de funciones cooperativas, **getch** entrega el siguiente carácter de la entrada que va a ser considerado; **ungetch** reintegra el carácter devuelto a la entrada, de modo que llamadas posteriores a **getch** lo regresarán antes de leer algo nuevo de la entrada.

Cómo trabajan juntas es sencillo, **ungetch** coloca el carácter regresado en un buffer compartido —un arreglo de caracteres. **getch** lee del buffer si hay algo allí y llama a **getchar** si el buffer está vacío. También debe existir una variable índice que registre la posición del carácter actual en el buffer temporal.

Puesto que el buffer y el índice son compartidos por **getch** y **ungetch** y deben retener sus valores entre llamadas, deben ser externos a ambas rutinas. Así, podemos escribir **getch**, **ungetch** y sus variables compartidas como:

```
#define BUFSIZE 100

char buf[BUFSIZE];   /* buffer para ungetch */
int bufp = 0;        /* siguiente posición libre en buf */

int getch(void) /* obtiene un (posiblemente ya regresado) carácter */
{
    return (bufp > 0) ? buf[--bufp] : getchar ( );
}
void ungetch(int c) /* regresa carácter a la entrada */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: demasiados caracteres\n");
    else
        buf[bufp++] = c;
}
```

La biblioteca estándar incluye una función **ungetc** que proporciona el regreso de un carácter; esto se verá en el capítulo 7. Se ha utilizado un arreglo para lo que se regresa a la entrada, en lugar de un carácter sencillo, para dar una idea más general.

Ejercicio 4-3. Dada la estructura básica, es fácil extender la calculadora. Agregue el operador módulo (%) y consideraciones para números negativos. □

Ejercicio 4-4. Agregue órdenes para imprimir el elemento al tope de la pila sin sacarlo de ella, para duplicarlo y para intercambiar los dos elementos del tope. Agregue una orden para limpiar la pila. □

Ejercicio 4-5. Agregue acceso a funciones de biblioteca como **sin**, **exp** y **pow**. Consulte **<math.h>** en el apéndice B, sección 4. □

Ejercicio 4-6. Agregue órdenes para manipular variables. (Es fácil proporcionar veintiséis variables con nombres de una letra.) Añada una variable para el valor impreso más reciente. □

Ejercicio 4-7. Escriba una rutina **ungets(s)** que regresa a la entrada una cadena completa. ¿Debe **ungets** conocer acerca de **buf** y **bufp**, o sólo debe usar **ungetch**? □

Ejercicio 4-8. Suponga que nunca existirá más de un carácter de regreso. Modifique **getch** y **ungetch** de acuerdo con eso. □

Ejercicio 4-9. Nuestros **getch** y **ungetch** no manejan correctamente un **EOF** que se regresa. Decida cuáles deben ser sus propiedades si se regresa un **EOF**, y después realice su diseño. □

Ejercicio 4-10. Una organización alternativa emplea **getline** para leer una línea completa de entrada; esto hace innecesarios a **getch** y a **ungetch**. Corrija la calculadora para que use este planteamiento. □

4.4 Reglas de alcance

Las funciones y variables externas que constituyen un programa en C no necesitan ser compiladas a la vez; el texto fuente del programa se puede tener en varios archivos y se pueden cargar rutinas previamente compiladas de biblioteca. Entre las preguntas de interés están

- ¿Cómo se escriben las declaraciones de modo que las variables sean declaradas adecuadamente durante la compilación?
- ¿Cómo se arreglan las declaraciones de modo que todas las piezas se conecten adecuadamente cuando se carga el programa?
- ¿Cómo se organizan las declaraciones de modo que sólo haya una copia?
- ¿Cómo se inicializan las variables externas?

Discutamos estos temas reorganizando el programa de la calculadora en varios archivos. En términos prácticos, la calculadora es demasiado pequeña para que

convenga separarla, pero es una excelente ilustración de los conceptos que surgen en programas mayores.

El *alcance* de un nombre es la parte del programa dentro del cual se puede usar el nombre. Para una variable automática declarada al principio de una función, el alcance es la función dentro de la cual está declarado el nombre. Las variables locales con el mismo nombre que estén en funciones diferentes no tienen relación. Lo mismo es válido para los parámetros de una función, que en efecto son variables locales.

El alcance de una variable o función externa abarca desde el punto en que se declara hasta el fin del archivo que se está compilando. Por ejemplo, si **main**, **sp**, **val**, **push**, y **pop** están definidas en un archivo, en el orden expuesto anteriormente, esto es,

```
main( ) {...}

int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

entonces las variables **sp** y **val** se pueden utilizar en **push** y **pop** simplemente nombrándolas; no se necesita ninguna otra declaración. Pero estos nombres no son visibles en **main**, ni **push** ni **pop**.

Por otro lado, si se va a hacer referencia a una variable externa antes de su definición, o si está definida en un archivo fuente diferente al que se está utilizando, entonces es obligatoria una declaración **extern**.

Es importante distinguir entre la *declaración* de una variable externa y su *definición*. Una declaración expone las propiedades de una variable (principalmente su tipo); una definición también provoca que reserve un espacio para almacenamiento. Si las líneas

```
int sp;
double val[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas **sp** y **val**, reservan un espacio para almacenamiento y también sirven como declaración para el resto de ese archivo fuente. Por otro lado, las líneas

```
extern in sp;
extern double val[ ];
```

declaran para el resto del archivo que **sp** es un **int** y que **val** es un arreglo **double** (cuyo tamaño se determina en algún otro lugar), pero no crean las variables ni les reservan espacio.

Sólo debe existir una *definición* de una variable externa entre todos los archivos que forman un programa fuente; otros archivos pueden contener declaraciones

extern para tener acceso a ellas. (También puede haber declaraciones **extern** en el archivo que contiene la definición.) Los tamaños de los arreglos deben ser especificados con la definición, pero es optativo hacerlo en una declaración **extern**.

La inicialización de una variable externa sólo va con su definición.

Aunque no es una organización idónea para este programa, las funciones push y pop pueden definirse en un archivo, y las variables val y sp podrían ser definidas e inicializadas en otro. Entonces se necesitarían las siguientes definiciones y declaraciones para enlazarlas:

En el archivo 1:

```
extern int sp;
extern double val[ ]

void push(double f) { ... }

double pop(void) { ... }
```

En el archivo 2:

```
int sp = 0
double val[MAXVAL];
```

Debido a que las declaraciones **extern** que se encuentran en el *archivo 1* están situadas antes y afuera de las definiciones de funciones, se aplican a todas las funciones; un conjunto de declaraciones basta para todo el *archivo 1*. Esta misma organización también sería necesaria si las definiciones de **sp** y **val** se encontraran después de su uso en un archivo.

4.5 Archivo de encabezamiento header

Consideremos ahora la división del programa de la calculadora en varios archivos fuente, como podría ser si cada uno de los componentes fuera sustancialmente mayor. La función **main** irá dentro de un archivo, al que llamaremos main.c; push, pop y sus variables van dentro de un segundo archivo, **stack.c**; getop va en un tercero, getop.c. Finalmente, **getch** y **ungetch** van dentro de un cuarto archivo, **getch.c**; las separaremos de las otras debido a que podrían venir de una biblioteca compilada separadamente en un programa realista.

Hay algo más de qué preocuparse —las definiciones y declaraciones compartidas entre archivos. Debemos centralizarlas hasta donde sea posible, de modo que haya sólo una copia por mantener mientras se desarrolla el programa. En consecuencia, situaremos este material común en un *archivo* tipo *header*, calc.h, que se incluirá donde sea necesario. (La línea **#include** se describe en la sección 4.11.) Entonces, el programa resultante se ve como sigue:

calc.h:

```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char [ ]);
int getch(void);
void ungetch(int);
```

main.c:

```
#include <stdio.h>
#include <math.h>
#include "calc.h"
#define MAXOP 100
main ( ) {
        ...
}
```

getop.c:

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop( ) {
        ...
}
```

stack.c:

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
        ...
}
double pop(void) {
        ...
}
```

getch.c:

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
        ...
}
void ungetch(int) {
        ...
}
```

Existe un compromiso entre el deseo de que cada archivo sólo tenga acceso a la información que necesita para su trabajo y la realidad práctica de que es más difícil mantener más archivos tipo *header*. Hasta un tamaño moderado de programa, probablemente es mejor tener un archivo de encabezamiento que contenga todo lo que será compartido entre las partes del programa; ésta es la decisión que tomamos aquí. Para un programa mucho más grande, se necesitaría más organización y más archivos tipo *header*.

4.6 Variables estáticas

Las variables sp y val en stack.c, y buf y bufp en getch.c, son para el uso privado de las funciones que están en sus respectivos archivos fuente, y se supone

que nada más tiene acceso a ellas. La declaración **static**, aplicada a una variable o función externa, limita el alcance de ese objeto al resto del archivo fuente que se está compilando. Así las variables **static** externas proporcionan una forma de ocultar nombres como **buf** y **bufp** en la combinación **getch-ungetch**, que deben ser externas para que puedan ser compartidas, aunque no deben ser visibles a los usuarios de **getch** y **ungetch**.

El almacenamiento estático se especifica al anteponer a la declaración normal la palabra **estatic**. Si las dos rutinas y las dos variables se compilan en un archivo, como en

```
static char buf[BUFSIZE];   /* buffer para ungetch */
static int bufp = 0;        /* siguiente posición libre en buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

entonces ninguna otra rutina será capaz de tener acceso a **buf** ni a **bufp**, y esos nombres no entrarán en conflicto con los mismos nombres que estén en otros archivos del mismo programa. De la misma manera, las variables que **push** y **pop** utilizan para la manipulación de la pila se pueden ocultar, declarando **sp** y **val** como **static**.

La declaración **static** externa se usa con más frecuencia en variables, pero también se puede aplicar a funciones. Normalmente, los nombres de funciones son globales, visibles a cualquier parte del programa completo. Sin embargo, si una función se declara como **static**, su nombre es invisible fuera del archivo en el que está declarada.

La declaración **static** también puede aplicarse a variables internas. Las variables internas **static** son locales a una función en particular, tal como lo son las variables automáticas, pero a diferencia de ellas, mantienen su existencia en lugar de ir y venir cada vez que se activa la función. Eso significa que las variables internas **static** proporcionan almacenamiento privado y permanente dentro de una función.

Ejercicio 4-11. Modifique **getop** de modo que no necesite utilizar **ungetch**. Sugerencia: emplee una variable **static** interna. □

4.7 Variables tipo registro

Una declaración **register** indica al compilador que la variable en cuestión se empleará intensamente. La idea es que las variables **register** se coloquen en registros de la máquina, lo que puede dar como resultado programas más pequeños y rápidos. Pero los compiladores tienen la libertad de ignorar esta sugerencia.

La declaración **register** se ve así:

```
register int x;
register char c;
```


etcétera. La declaración **register** sólo se puede aplicar a variables automáticas y a los parámetros formales de una función. En este último caso, aparece como

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

En la práctica existen restricciones en las variables tipo registro, que reflejan la realidad del equipo donde se opera. Sólo algunas variables de cada función se pueden mantener en registros, y sólo se permiten ciertos tipos. Sin embargo, el exceso de declaraciones tipo registro no provoca daños, puesto que la palabra **register** se ignora en las declaraciones excesivas o no permitidas. Además, no es posible tomar la dirección de una variable de tipo registro (tema que se tratará en el capítulo 5), sin importar si la variable está o no realmente en un registro. Las restricciones específicas sobre el número y tipo de estas variables varían de una máquina a otra.

4.8 Estructura de bloques

C no es un lenguaje estructurado en bloques en el sentido de Pascal o lenguajes semejantes, puesto que las funciones no se pueden definir dentro de otras funciones. Por otra parte, las variables se pueden definir en una modalidad de estructura de bloques dentro de una función. Las declaraciones de variables (incluyendo la inicialización) pueden seguir a la llave izquierda que indica *cualquier* proposición compuesta, no sólo la que inicia a una función. Las variables declaradas de esta manera ocultan cualquier nombre idéntico de variables en bloques externos, y permanecen hasta que se encuentra la llave derecha que corresponde con la inicial. Por ejemplo, en

```
if (n > 0) {
    int i; /* declara una nueva i */

    for (i = 0; i < n; i++)
        ...
}
```

el alcance de la variable **i** es la rama "verdadera" del **if**; esta **i** no tiene nada que ver con alguna **i** fuera del bloque. Una variable automática declarada e inicializada en un bloque se inicializa cada vez que se entra al bloque. Una variable **static** se inicializa sólo la primera vez que se entra al bloque.

Las variables automáticas, incluyendo los parámetros formales, también esconden a las variables y funciones externas del mismo nombre. Dadas las declaraciones

```
int x;
int y;

f (double x)
{
      double y;
      ...
}
```

en la función **f**, las ocurrencias de **x** se refieren al parámetro, que es un **double**; fuera de **f**, se refieren al **int** externo. Lo mismo es válido para la variable **y**.

Por estilo, es mejor evitar nombres de variables que coinciden con nombres de un alcance exterior; la potencialidad de confusión y error es muy grande.

4.9 Inicialización

La inicialización ya se ha mencionado muchas veces, pero siempre alrededor de algún otro tema. Esta sección resume algunas de las reglas, ahora que ya se han discutido las diferentes categorías de almacenamiento.

En ausencia de una inicialización explícita, se garantiza que las variables externas y estáticas se inicializarán en cero; las variables automáticas y tipo registro tienen valores iniciales indefinidos (esto es, basura).

Las variables escalares se pueden inicializar cuando se definen, siguiendo al nombre con un signo de igual y una expresión:

```
int x = 1;
char apóstrofo = '\'';
long día = 1000L * 60L * 60L * 24L;   /* milisegundos/día */
```

Para variables externas y estáticas, el inicializador debe ser una expresión constante; la inicialización se realiza una vez, conceptualmente antes de que el programa inicie su ejecución. Para variables automáticas y tipo registro, se hace cada vez que se entra a la función o bloque.

Para variables automáticas y tipo registro, el inicializador no se limita a una constante: puede ser cualquier expresión que contenga valores previamente definidos, incluso llamadas a funciones. Por ejemplo, la inicialización del programa de búsqueda binaria de la sección 3.3 podría escribirse como

```
int binsearch(int x, int v [ ], int n)
{
      int low = 0;
      int high = n - 1;
      int mid;
      ...
}
```


en lugar de

```
int low, high, mid;

low = 0;
high = n - 1;
```

En efecto, las inicializaciones de variables automáticas son sólo abreviaturas de proposiciones de asignación. La elección es en gran medida cuestión de gusto. Nosotros hemos empleado generalmente asignaciones explícitas, debido a que los inicializadores en las declaraciones son difíciles de ver y lejanos del lugar de uso.

Un arreglo puede ser inicializado al seguir su declaración con una lista de inicializadores encerrados entre llaves y separados por comas. Por ejemplo, para inicializar un arreglo **días** con el número de días de cada mes:

```
int días [ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Cuando se omite el tamaño de un arreglo, el compilador calculará la longitud contando los inicializadores, los cuales son 12 en este caso.

Si existen menos inicializadores para un arreglo que los del tamaño especificado, los otros serán cero para variables externas o estáticas, pero basura para automáticas. Es un error tener demasiados inicializadores. No hay forma de indicar la repetición de un inicializador, ni de inicializar un elemento que está a la mitad de un arreglo sin proporcionar también todos los valores precedentes.

Los arreglos de caracteres son un caso especial de inicialización; se puede utilizar una cadena en lugar de la notación de llaves y comas:

```
char patrón[ ] = "ould";
```

es más corto pero equivalente a

```
char patrón[ ] = { 'o', 'u', 'l', 'd', '\0' };
```

En este caso, el tamaño del arreglo es cinco (cuatro caracteres más el terminador '\0').

4.10 Recursividad

Las funciones de C pueden emplearse recursivamente; esto es, una función puede llamarse a sí misma ya sea directa o indirectamente. Considere la impresión de un número como una cadena de caracteres. Como ya se mencionó anteriormente, los dígitos se generan en orden incorrecto: los dígitos de orden inferior están disponibles antes de los dígitos de orden superior, pero se deben imprimir en el orden invertido.

Existen dos soluciones a este problema. Una es almacenar los dígitos en un arreglo tal como se generan, y después imprimirlos en orden inverso, como se hizo

con itoa en la sección 3.6. La alternativa es una solución recursiva, en la que printd primero se llama a sí misma para tratar con los primeros dígitos, y después imprime el dígito del final. De nuevo, esta versión puede fallar con el número negativo más grande.

```
#include <stdio.h>

/* printd: imprime n en decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10);
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

Cuando una función se llama a sí misma recursivamente, cada invocación obtiene un conjunto nuevo de todas las variables automáticas, independiente del conjunto previo. Así, en printd(123) el primer printd recibe el argumento n = 123. Pasa 12 al segundo printd, que a su vez pasa 1 a un tercero. El printd del tercer nivel imprime 1, después regresa al segundo nivel. Ese printd imprime 2, después regresa al primer nivel. Ese imprime 3 y termina.

Otro buen ejemplo de recursividad es *quicksort*, un algoritmo de ordenamiento desarrollado en 1962 por C. A. R. Hoare. Dado un arreglo, un elemento se selecciona y los otros se particionan en dos subconjuntos —aquellos menores que el elemento de la partición y aquellos mayores o iguales a él. El mismo proceso se aplica después recursivamente a los dos subconjuntos. Cuando un subconjunto tiene menos de dos elementos no necesita ya de ningún ordenamiento; esto detiene la recursividad.

Nuestra versión de *quicksort* no es la más rápida posible, pero es una de las más simples. Empleamos el elemento intermedio de cada subarreglo para particionar.

```
/* qsort: ordena v[left]...v[right] en orden ascendente */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[ ], int i, int j);

    if (left >= right)          /* no hace nada si el arreglo contiene */
        return;                 /* menos de dos elementos */
```



```
    swap(v, left, (left + right)/2);  /* mueve el elemento de partición */
    last = left;                      /* a v[0] */
    for (i = left+1; 1 <= right; i++)    /* partición */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);              /* regresa el elemento de partición */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Pasamos la operación de intercambio a una función separada swap, puesto que ocurre tres veces en qsort.

```
/* swap: intercambia v[i] y v [j] */
void swap(int v[ ], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

La biblioteca estándar incluye una versión de qsort que puede ordenar objetos de cualquier tipo.

La recursividad no puede proporcionar un ahorro en almacenamiento, puesto que en algún lugar se debe mantener una pila de los valores procesados. Ni será más rápida. Pero el código recursivo es más compacto y frecuentemente mucho más fácil de escribir y de entender que su equivalente no recursivo. La recursividad es especialmente conveniente para estructuras de datos definidas en forma recursiva, como árboles; veremos un agradable ejemplo en la sección 6.5.

Ejercicio 4-12. Adapte las ideas de printd al escribir la versión recursiva del programa itoa; esto es, convierta un entero en una cadena llamando a una rutina recursiva. □

Ejercicio 4-13. Escriba una versión recursiva de la función reverse(s), que invierte la cadena s en su lugar. □

4.11 El preprocesador de C

C proporciona ciertas facilidades de lenguaje por medio de un preprocesador, que conceptualmente es un primer paso separado en la compilación. Los dos elementos que se usan con más frecuencia son #include, para incluir el contenido de un archivo durante la compilación, y **#define**, para reemplazar un símbolo por

una secuencia arbitraria de caracteres. Otras características que se describen en esta sección incluyen compilación condicional y macros con argumentos.

4.11.1 Inclusión de archivos

La inclusión de archivos facilita el manejo de grupos de #define y declaraciones (entre otras cosas). Cualquier línea fuente de la forma

```
#include "nombre"
```

o

```
#include <nombre>
```

se reemplaza por el contenido del archivo *nombre*. Si el *nombre* se encierra entre comillas, la búsqueda del archivo comienza normalmente donde se encontró el programa fuente; si no se encuentra allí, o si el nombre se delimita por < y >, la búsqueda sigue una ruta predefinida por la implantación para encontrar el archivo. Un archivo incluido puede contener líneas #include.

Frecuentemente existen varias líneas #include al principio de un archivo fuente, para incluir proposiciones #define y declaraciones extern comunes, o para tener acceso a la declaración del prototipo de una función de biblioteca desde *headers* como < stdio.h >. (Estrictamente hablando, no necesitan ser archivos; los detalles de cómo se tiene acceso a los *headers* dependen de la implantación.)

#include es la mejor manera de enlazar las declaraciones para un programa grande. Garantiza que todos los archivos fuente se suplirán con las mismas definiciones y declaraciones de variables, y así elimina un tipo de error particularmente desagradable. Por supuesto, cuando se cambia un archivo include, se deben recompilar todos los archivos que dependen de él.

4.11.2 Substitución de macros

Una definición tiene la forma

```
#define   nombre   texto de reemplazo
```

Pide la substitución de una macro del tipo más sencillo —las siguientes ocurrencias de *nombre* serán substituidas por el *texto de reemplazo*. El nombre en un #define tiene la misma forma que un nombre de variable; el texto de reemplazo es arbitrario. Normalmente el texto de reemplazo es el resto de la línea, pero una definición extensa puede continuarse en varias líneas, colocando una \ al final de cada línea que va a continuar. El alcance de un nombre definido con #define va desde su punto de definición hasta el fin del archivo fuente que se compila. Una definición puede emplear definiciones previas. Las substituciones se realizan sólo para elementos, y no sucede dentro de cadenas delimitadas por comillas. Por ejemplo, si **AFIRMA** es un nombre definido, no habrá substitución en printf ("AFIRMA") ni en AFIRMATIVO.

Cualquier nombre puede definirse con cualquier texto de reemplazo. Por ejemplo.

```
#define  porsiempre  for (;;) /* ciclo infinito */
```

define una nueva palabra, porsiempre, para un ciclo infinito.

También es posible definir macros con argumentos, para que el texto de reemplazo pueda ser diferente para diferentes llamadas de la macro. Como un ejemplo, defina una macro llamada max;

```
#define max (A, B) ((A) > (B) ? (A) : (B))
```

Aunque aparenta ser una llamada a función, el uso de **max** se expande a código. Cada ocurrencia de un parámetro formal (**A** o **B**) será reemplazada por el argumento real correspondiente. Así, la línea

```
x = max(p+q, r+s);
```

será reemplazada por la línea

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

En tanto que los argumentos se traten consistentemente, esta macro servirá para cualquier tipo de datos; no hay necesidad de diferentes tipos de **max** para diferentes tipos de datos, como la habría con las funciones.

Si examina la expansión de **max**, notará algunos riesgos. Las expresiones se evalúan dos veces; esto es malo si involucra efectos colaterales como operadores incrementales o de entrada y salida. Por ejemplo,

```
max(i++, j++)     /* INCORRECTO */
```

incrementará el valor más grande dos veces. También debe tenerse algún cuidado con los paréntesis, para asegurar que se preserva el orden de evaluación; considere qué pasa cuando la macro

```
#define cuadrado(x)  x * x     /* INCORRECTO */
```

se invoca como cuadrado (z + 1).

Sin embargo, las macros son valiosas. Un ejemplo práctico viene de <stdio.h>, donde getchar y putchar se definen frecuentemente como macros para evitar el exceso de tiempo de ejecución de una llamada a función por cada carácter procesado. Las funciones en <ctype.h> también se realizan generalmente como macros.

Los nombres se pueden hacer indefinidos con #undef, para asegurar que una rutina es realmente una función, no una macro:

```
#undef getchar
int getchar (void) { ... }
```

Los parámetros formales no se reemplazan dentro de cadenas entre comillas. Sin embargo, si un nombre de parámetro está precedido por un # en el texto de reemplazo, la combinación se expandirá en una cadena entre comillas, con el parámetro reemplazado por el argumento real. Esto puede combinarse con concatenación de cadenas para hacer, por ejemplo, una macro de impresión para depuración:

```
#define   dprint(expr)   printf(#expr " = %g\n", expr)
```

Cuando se invoca, como en

```
dprint(x/y);
```

la macro se expande en

```
printf("x/y" " = %g\n", x/y);
```

y las cadenas se concatenan, así el efecto es

```
printf("x/y = %g\n", x/y);
```

Dentro del argumento real, cada " se reemplaza por \" y cada \ por \\, así que el resultado es una constante de cadena legítima.

El operador ## del preprocesador proporciona una forma de concatenar argumentos reales durante la expansión de una macro. Si un parámetro que está en el texto de reemplazo es adyacente a un ##, es reemplazado por el argumento real, se eliminan el ## y los espacios en blanco que lo rodean, y el resultado se rastrea de nuevo. Por ejemplo, la macro **paste** concatena sus dos argumentos:

```
#define paste(front, back) front ## back
```

así, **paste(nombre, 1)** crea el token **nombre1**.

Las reglas para el uso anidado de ## son misteriosas; en el apéndice A se pueden encontrar mayores detalles.

Ejercicio 4-14. Defina una macro swap(t,x,y) que intercambie dos argumentos de tipo t. (La estructura de bloques ayudará.) □

4.11.3 Inclusión condicional

Es posible controlar el preprocesamiento mismo con proposiciones condicionales que se evalúan durante esa etapa. Esto proporciona una forma de incluir código selectivamente, dependiendo del valor de condiciones evaluadas durante la compilación.

La línea **#if** evalúa una expresión constante entera (que no puede incluir **sizeof**, *casts* o constantes **enum**). Si la expresión es diferente de cero, se incluyen las siguientes líneas hasta un **#endif**, **#elif** o **#else**. (La proposición de procesador **#elif** es como **else if**). La expresión **defined**(*nombre*) en un **#if** es 1 si el *nombre* se ha definido, y 0 de otra manera.

Por ejemplo, para asegurarse de que el contenido de un archivo **hdr.h** se incluya sólo una vez, el contenido del archivo se delimita con una condicional como ésta:

```
#if !defined (HDR)
#define HDR

/* el contenido de hdr.h va aquí */

#endif
```

La primera inclusión de **hdr.h** define el nombre **HDR**; las siguientes inclusiones encontrarán definido al nombre y pasarán hacia el **#endif**. Un estilo semejante puede emplearse para evitar incluir archivos varias veces. Si este estilo se utiliza en forma consistente, entonces cada *header* puede en sí mismo incluir cualquier otro del que dependa, sin que el usuario tenga que tratar con la interdependencia.

La siguiente secuencia prueba el nombre **SYSTEM** para decidir cuál versión de un *header* incluir:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```

Las líneas **#ifdef** e **#ifndef** son formas especializadas que prueban si un nombre está definido. El primer ejemplo de **#if** de más arriba pudo haberse escrito

```
#ifndef HDR
#define HDR

/* contenido de hdr.h va aquí */

#endif
```

CAPITULO 5: **Apuntadores y arreglos**

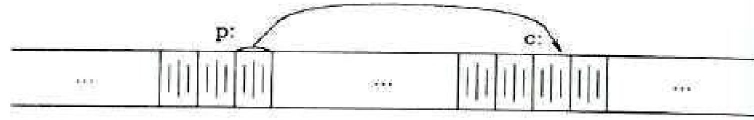
Un apuntador es una variable que contiene la dirección de una variable. Los apuntadores se utilizan mucho en C, en parte debido a que ellos son en ocasiones la única forma de expresar una operación, y en parte debido a que por lo general llevan un código más compacto y eficiente de lo que se puede obtener en otras formas. Los apuntadores y los arreglos están relacionados íntimamente; este capítulo también explora estas relaciones y muestra cómo explotarlas.

Los apuntadores se han puesto junto a la proposición goto como una forma maravillosa de crear programas ininteligibles. Esto es verdadero cuando se utilizan en forma descuidada, y es fácil crear apuntadores que señalen a algún lugar inesperado. Sin embargo, con disciplina, los apuntadores pueden también emplearse para obtener claridad y simplicidad. Este es el aspecto que trataremos de ilustrar.

El principal cambio en ANSI C es hacer explícitas las reglas acerca de cómo pueden manipularse los apuntadores, obligando a lo que los buenos programadores ya practican y lo que los buenos compiladores ya imponen. Además, el tipo void * (apuntador a **void**) reemplaza a char * como el tipo apropiado para un apuntador genérico.

5.1 Apuntadores y direcciones

Empecemos con un dibujo simplificado de cómo se organiza la memoria. Una máquina típica tiene un arreglo de celdas de memoria numeradas o direccionadas consecutivamente, que pueden manipularse individualmente o en grupos contiguos. Una situación común es que cualquier byte puede ser un char, un par de celdas de un byte pueden tratarse como un entero **short**, y cuatro bytes adyacentes forman un **long**. Un apuntador es un grupo de celdas (generalmente dos o cuatro) que pueden mantener una dirección. Así, si c es un char y p es un apuntador que apunta a él, podría representarse la situación de esta manera:



El operador unario & da la dirección de un objeto, de modo que la proposición.

```
p = &c;
```

asigna la dirección de c a la variable p, y se dice que p "apunta a" c. El operador & sólo se aplica a objetos que están en memoria: variables y elementos de arreglos. No puede aplicarse a expresiones, constantes o variables tipo registro.

El operador unario * es el operador de *indirección* o *desreferencia*; cuando se aplica a un apuntador, da acceso al objeto al que señala el apuntador. Supóngase que x y y son enteros e ip es un apuntador a **int**. Esta secuencia artificial muestra cómo declarar un apuntador y cómo emplear & y *:

```
int x = 1, y = 2, z[10];
int *ip;         /* ip es un apuntador a int */

ip = &x;         /* ip ahora apunta a x */
y = *ip;         /* y es ahora 1 */
*ip = 0;         /* x es ahora 0 */
ip = &z[0];      /* ip ahora apunta a z[0] */
```

Las declaraciones de **x**, **y** y **z** son lo que hemos visto todo el tiempo. La declaración del apuntador ip,

```
int *ip;
```

funciona como mnemónico; dice que la expresión *ip es un int. La sintaxis de la declaración para una variable imita la sintaxis de expresiones en las que la variable puede aparecer. Este razonamiento se aplica también a la declaración de funciones. Por ejemplo,

```
double *dp, atof(char *);
```

indica que en una expresión *dp y atof(s) tienen valores de tipo **double**, y que el argumento de atof es un apuntador a char.

También se debe notar la implicación que tiene el hecho de que un apuntador está restringido a señalar a una clase particular de objeto: cada apuntador señala a un tipo específico de datos. (Hay una excepción: un "apuntador a void" se emplea para mantener cualquier tipo de apuntador, pero en sí mismo no puede ser *desreferenciado*. Esto se volverá a tratar en la sección 5.11.)

Si ip apunta al entero **x**, entonces *ip puede presentarse en cualquier contexto donde **x** pueda hacerlo, así que

```
*ip = *ip + 10;
```

incrementa *ip en 10.

Los operadores unarios * y & se ligan más estrechamente que los operadores aritméticos; así, la asignación

```
y = *ip + 1
```

toma aquello a lo que apunte **ip**, le agrega 1, y asigna el resultado a y, mientras que

```
*ip += 1
```

incrementa en uno aquello a que **ip** apunta, como lo hace

```
++*ip
```

y

```
(*ip)++
```

Los paréntesis son necesarios en este último ejemplo; sin ellos, la expresión incrementaría **ip** en lugar de a lo que apunta, debido a que los operadores unarios como * y ++ se asocian de derecha a izquierda.

Por último, puesto que los apuntadores son variables, se pueden emplear sin *desreferenciamiento*. Por ejemplo, si iq es otro apuntador a int,

```
iq = ip
```

copia el contenido de ip en iq; así, hace que iq apunte a lo que ip está apuntando.

5.2 Apuntadores y argumentos de funciones

Puesto que C pasa los argumentos de funciones por valor, no existe una forma directa para que la función que se invoca altere una variable de la función que la llama. Por ejemplo, una rutina de ordenamiento podría intercambiar dos elementos desordenados con una función llamada swap. No es suficiente escribir

```
swap(a, b);
```

donde la función swap está definida como

```
void swap(int x, int y) /* INCORRECTO */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Debido a la llamada por valor, swap no puede afectar los argumentos a y b que están en la rutina que la llamó. La función anterior sólo intercambia *copias* de a y de b.

La forma de obtener los resultados que se desean es que el programa invocador pase *apuntadores* a los valores que se cambiarán:

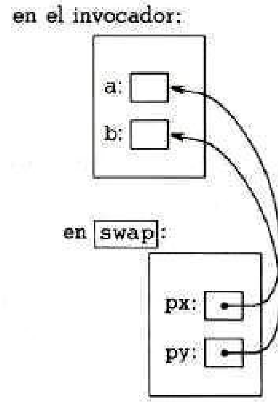
```
swap(&a, &b);
```

Puesto que el operador & produce la dirección de una variable, &a es un apuntador a a. Dentro de la misma función swap, los parámetros se declaran para ser apuntadores, y se tiene acceso a los operandos indirectamente a través de ellos.

```
void swap(int *px, int *py)    /* intercambia *px y *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Gráficamente:



Los argumentos tipo apuntador permiten a una función tener acceso y cambiar objetos que están en la función que la llamó. Como ejemplo, considere una función getint que realiza una conversión de entrada en formato libre, desglosando un flujo de caracteres en valores enteros, un entero por llamada. Así, getint tiene que regresar el valor encontrado y también una señal de fin de archivo cuando ya no hay más que tomar. Esos valores tienen que regresarse por rutas separadas, para que sin importar qué valor se emplea para EOF, también pueda ser el valor de un entero de la entrada.

Una solución es hacer que getint regrese el estado de fin de archivo como su valor de función, usando un argumento apuntador para almacenar el entero convertido y tenerlo en la función invocadora. Este esquema también es utilizado por scanf, como se verá en la sección 7.4.

El siguiente ciclo llena un arreglo con enteros por medio de llamadas a getint:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

Cada llamada pone en array[n] el siguiente entero que se encuentra a la entrada e incrementa n. Obsérvese que es esencial pasar la dirección de array[n] a getint. De otra manera no hay forma de que getint comunique el entero convertido hacia la función invocadora.

Esta versión de getint regresa EOF como fin de archivo, cero si la siguiente entrada no es un número, y un valor positivo si la entrada contiene un número válido.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: obtiene el siguiente entero de la entrada y lo asigna a *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch()))    /* ignora espacios en blanco */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);    /* no es un número */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch( );
    for (*pn = 0; isdigit(c); c = getch( ))
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

A lo largo de getint, *pn se emplea como una variable int ordinaria. También se utilizó getch y ungetch (descritas en la sección 4.3) para que el carácter extra que debe leerse puede regresar a la entrada.

Ejercicio 5-1. Como se escribió, getint trata a un + o un − no seguido por un dígito como una representación válida de cero. Corríjala para que regrese tal carácter a la entrada. □

Ejercicio 5-2. Escriba getfloat, la analogía de punto flotante de getint. ¿Qué tipo regresa getfloat como su valor de función? □

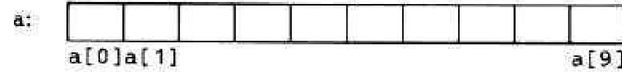
5.3 Apuntadores y arreglos

En C existe una fuerte relación entre apuntadores y arreglos, tan fuerte que deben discutirse simultáneamente. Cualquier operación que pueda lograrse por indexación de un arreglo también puede realizarse con apuntadores. La versión con apuntadores será por lo general más rápida, pero, al menos para los no iniciados, algo más difícil de entender.

La declaración

```
int a[10];
```

define un arreglo a de tamaño 10, esto es, un bloque de 10 objetos consecutivos llamados a[0],a[1],...,a[9].



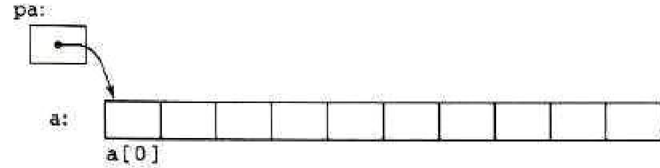
La notación a[i] se refiere al i-ésimo elemento del arreglo. Si pa es un apuntador a un entero, declarado como

```
int *pa;
```

entonces la asignación

```
pa = &a[0];
```

hace que pa apunte al elemento cero de a; esto es, pa contiene la dirección de a[0].



Ahora la asignación

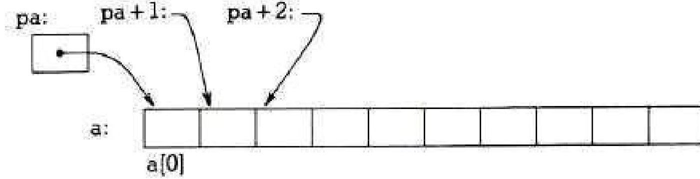
```
x = *pa;
```

copiará el contenido de a[0] en x.

Si pa apunta a un elemento en particular de un arreglo, entonces por definición pa+1 apunta al siguiente elemento, pa+i apunta i elementos después de pa, y pa−i apunta i elementos antes. Así, si pa apunta a a[0],

```
*(pa+1)
```

se refiere al contenido de a[1], pa+i es la dirección de a[i] y *(pa+i) es el contenido de a[i].



Lo anterior es verdadero sin importar el tipo o tamaño de las variables del arreglo a. El significado de "agregar 1 a un apuntador", y por extensión, toda la aritmética de apuntadores, es que pa+1 apunta al siguiente objeto, y pa+i apunta al i-ésimo objeto adelante de pa.

La correspondencia entre indexación y aritmética de apuntadores es muy estrecha. Por definición, el valor de una variable o expresión de tipo arreglo es la dirección del elemento cero del arreglo. Así, que después de la asignación

```
pa = &a[0];
```

pa y a tienen valores idénticos. Puesto que el nombre de un arreglo es un sinónimo para la localidad del elemento inicial, la asignación pa=&a[0] puede escribirse también como

```
pa = a;
```

Más sorprendente, al menos a primera vista, es el hecho de que una referencia a a[i] también puede escribirse como *(a+i). Al evaluar a[i], C la convierte inmediatamente a *(a+i); las dos formas son equivalentes. Al aplicar el operador & a ambas partes de esta equivalencia, se deriva que &a[i] y a+i también son idénticas: a+i es la dirección del i-ésimo elemento delante de a. Por otra parte, si pa es un apuntador, las expresiones pueden usarlo con un subíndice; pa[i] es idéntico a *(pa+i). En resumen, cualquier expresión de arreglo e índice es equivalente a una expresión escrita como un apuntador y un desplazamiento.

Existe una diferencia entre un nombre de arreglo y un apuntador, que debe tenerse en mente. Un apuntador es una variable, por esto pa=a y pa++ son legales. Pero un nombre de arreglo no es una variable; construcciones como a=pa y a++ son ilegales.

Cuando un nombre de arreglo se pasa a una función, lo que se pasa es la localidad del elemento inicial. Dentro de la función que se llama, este argumento es una variable local, y por lo tanto, un parámetro de nombre de arreglo es un apuntador, esto es, una variable que contiene una dirección. Se puede utilizar este hecho para escribir otra versión de **strlen**, que calcula la longitud de una cadena.

```
/* strlen: regresa la longitud de la cadena s */
int strlen(char *s)
{
     int n;

     for (n = 0; *s != '\0'; s++)
          n++;
     return n;
}
```

Puesto que s es un apuntador, es perfectamente legal incrementarlo; s++ no tiene efecto alguno sobre la cadena de caracteres de la función que llamó a **strlen**, sino que simplemente incrementa la copia privada del apuntador de strlen. Eso significa que llamadas como

```
strlen("hola, mundo");      /* constante de cadena */
strlen(array);              /* char array[100]; */
strlen(ptr);                /* char *ptr; */
```

sí funcionan.

Puesto que los parámetros formales en una definición de función,

```
char s[ ];
```

y

```
char *s;
```

son equivalentes, preferimos el último, porque indica más explícitamente que el parámetro es un apuntador. Cuando un nombre de arreglo se pasa a una función, ésta puede interpretar a su conveniencia que se ha manejado un arreglo o un apuntador, y manipularlo en consecuencia. Puede incluso emplear ambas notaciones si ello lo hace apropiado y claro.

Es posible pasar parte de un arreglo a una función, pasando un apuntador al inicio del subarreglo. Por ejemplo, si a es un arreglo,

```
f(&a[2])
```

y

```
f(a+2)
```


ambas pasan a la función f la dirección del subarreglo que inicia en a[2]. Dentro de f, la declaración de parámetros puede ser

```
f(int arr[ ]) { ... }
```

o

```
f(int *arr) { ... }
```

Así, hasta donde a f le concierne, el hecho de que el parámetro se refiera a parte de un arreglo más grande no es de consecuencia.

Si se está seguro de que los elementos existen, también es posible indexar hacia atrás en un arreglo; p[-1], p[-2], etc., son legítimos desde el punto de vista sintáctico, y se refieren a elementos que preceden inmediatamente a p[0]. Por supuesto, es ilegal hacer referencia a objetos que no estén dentro de los límites del arreglo.

5.4 Aritmética de direcciones

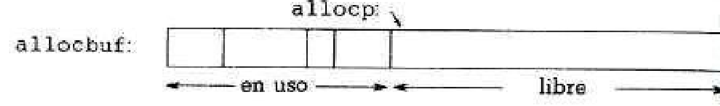
Si p es un apuntador a algún elemento de un arreglo, entonces p++ incrementa p para apuntar al siguiente elemento, y p+=i la incrementa para apuntar i elementos adelante de donde actualmente lo hace. Esas y otras construcciones semejantes son las formas más simples de aritmética de apuntadores o de direcciones.

El lenguaje C es consistente y regular en su enfoque a la aritmética de direcciones; su integración de apuntadores, arreglos y aritmética de direcciones es uno de los aspectos que le dan fuerza. Lo ilustraremos al escribir un rudimentario asignador de memoria. Hay dos rutinas: la primera, **alloc(n)**, regresa un apuntador p a n posiciones consecutivas, que pueden ser empleadas por el invocador de alloc para almacenar caracteres. La segunda, **afree(p)**, libera el almacenamiento adquirido en esta forma, de modo que pueda ser reutilizado posteriormente. Las rutinas son rudimentarias, puesto que las llamadas a **afree** deben realizarse en el orden opuesto a las llamadas realizadas a **alloc**. Es decir, el almacenamiento manejado por alloc y afree es una pila o lista del tipo último-que entra, primero-que-sale. La biblioteca estándar proporciona funciones análogas llamadas malloc y free que no tienen tales restricciones; en la sección 8.7 se mostrará cómo se pueden realizar.

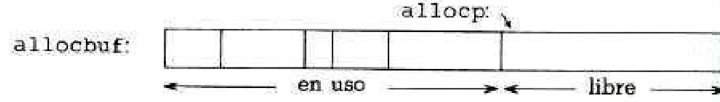
La implantación más sencilla es hacer que alloc maneje piezas de un gran arreglo de caracteres al que llamaremos **allocbuf**. Este arreglo está reservado para alloc y para **afree**. Puesto que éstas hacen su trabajo con apuntadores, no con índices, ninguna otra rutina necesita conocer el nombre del arreglo, el cual puede ser declarado como **static** en el archivo fuente que contiene a **alloc** y a **afree**, y así ser invisible hacia afuera. En la implantación práctica, el arreglo puede incluso no tener un nombre; podría obtenerse llamando a **malloc** o pidiendo al sistema operativo un apuntador hacia algún bloque sin nombre de memoria.

La otra información necesaria es cuánto de **allocbuf** se ha utilizado. Empleamos un apuntador, llamado **allocp**, que apunta hacia el siguiente elemento libre. Cuando se requieren **n** caracteres a **alloc**, primero revisa si hay suficiente espacio libre en **allocbuf**. Si lo hay, **alloc** regresa el valor actual de **allocp** (esto es, el principio del bloque libre), después lo incrementa en **n** para apuntar a la siguiente área libre. Si no hay espacio, **alloc** regresa cero, en tanto que **afree(p)** simplemente hace **allocp** igual a **p** si **p** está dentro de **allocbuf**.

antes de llamar a alloc:



después de llamar a alloc:




```
#define ALLOCSIZE 10000 /* tamaño del espacio disponible */

static char allocbuf[ALLOCSIZE];   /* almacenamiento para alloc */
static char *allocp = allocbuf;    /* siguiente posición libre */

char *alloc(int n)    /* regresa un apuntador a n caracteres */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* si cabe */
        allocp += n;
        return allocp - n; /* antigua p */
    } else        /* no hay suficiente espacio */
        return 0;
}
void afree(char *p)   /* almacenamiento libre apuntado por p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

En general, un apuntador puede ser inicializado tal como cualquier otra variable, aunque normalmente los únicos valores significativos son cero o una expresión que involucre la dirección de un dato previamente definido y de un tipo apropiado. La declaración

```
static char *allocp = allocbuf;
```


define a **allocp** como un apuntador a caracteres y lo inicializa para apuntar al principio de **allocbuf**, que es la siguiente posición libre cuando el programa comienza. Esto también podría haberse escrito

```
static char *allocp = &allocbuf[0];
```

puesto que el nombre del arreglo *es* la dirección del elemento cero-ésimo.

La prueba

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* si cabe */
```

comprueba si existe suficiente espacio para satisfacer la petición de **n** caracteres. Si lo hay, el nuevo valor de **allocp** sería, cuando mucho, uno adelante del fin de **allocbuf**. Si la petición puede satisfacerse, **alloc** regresa un apuntador al principio de un bloque de caracteres (nótese la declaración de la función). De lo contrario, **alloc** debe regresar alguna señal de que no queda espacio. El lenguaje C garantiza que cero nunca es una dirección válida para datos y por lo tanto puede usarse un valor de cero como retorno para señalar un suceso anormal, en este caso, falta de espacio.

Los apuntadores y los enteros no son intercambiables. Cero es la única excepción: la constante cero puede ser asignada a un apuntador, y éste puede compararse contra la constante cero. La constante simbólica **NULL** se emplea con frecuencia en lugar de cero, como un mnemónico para indicar más claramente que es un valor especial para un apuntador. **NULL** está definido en **<stdio.h>**. De aquí en adelante se utilizará **NULL**.

Pruebas como

```
if (allocbuf + ALLOCSIZE - allocp >= n) { /* si cabe */
```

y

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

muestran varias facetas importantes de la aritmética de apuntadores. Primero, los apuntadores pueden compararse bajo ciertas circunstancias. Si **p** y **q** apuntan a miembros del mismo arreglo, entonces relaciones como **==**, **!=**, **<**, **>=**, etc., funcionan correctamente. Por ejemplo,

```
p < q
```

es verdadero si **p** apunta a un elemento que está antes en el arreglo de lo que está al que apunta **q**. Cualquier apuntador puede ser comparado por su igualdad o desigualdad con cero. Pero está indefinido el comportamiento para la aritmética o comparaciones con apuntadores que no apuntan a miembros del mismo arreglo. (Existe una excepción: la dirección del primer elemento que está después del fin de un arreglo puede emplearse en aritmética de apuntadores.)

Segundo, ya se ha observado que un apuntador y un entero pueden sumarse o restarse. La construcción

```
p + n
```

significa la dirección del n-ésimo objeto adelante del que apunta actualmente p. Esto es verdadero sin importar la clase de objeto al que apunta p; n es escalada de acuerdo con el tamaño de los objetos a los que apunta p, lo cual está determinado por la declaración de p. Si un int es de cuatro bytes, por ejemplo, la escala para el int será de cuatro.

La resta de apuntadores también es válida: si p y q apuntan a elementos del mismo arreglo, y p<q, entonces q−p+1 es el número de elementos desde p hasta q, inclusive. Este hecho puede usarse para escribir todavía otra versión de **strlen**:

```
/* strlen: regresa la longitud de la cadena s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

En su declaración, p se inicializa en s, esto es, para apuntar al primer carácter de la cadena. En el ciclo **while**, cada carácter se examina en su turno hasta que al final se encuentra el '\0'. Debido a que p apunta a caracteres, p++ avanza p al siguiente carácter cada vez, y p−s da el número de caracteres que se avanzaron, esto es, la longitud de la cadena. (El número de caracteres en la cadena puede ser demasiado grande como para almacenarse en un int. El *header* <stddef.h> define un tipo **ptrdiff_t**, que es suficientemente grande para almacenar la diferencia signada de dos valores apuntadores. Sin embargo, si se es muy cauteloso, se debe usar **size_t** para el tipo de retorno de **strlen**, para coincidir con la versión de la biblioteca estándar, **size_t** es el tipo de entero sin signo que regresa el operador **sizeof**.)

La aritmética de apuntadores es consistente: si estuviéramos tratando con floats, que ocupan más espacio de memoria que los chars, y si p fuera un apuntador a float, p++ avanzaría al siguiente float. Así, podemos escribir otra versión de alloc que mantenga floats en lugar de chars, simplemente cambiando de char a float en todo alloc y afree. Todas las manipulaciones de apuntadores tomarán automáticamente en cuenta el tamaño de los objetos apuntados.

Las operaciones válidas de apuntadores son asignación de apuntadores del mismo tipo, suma y substracción de un apuntador y un entero, resta o comparación de dos apuntadores a miembros del mismo arreglo, y asignación o comparación con cero. Toda otra aritmética de apuntadores es ilegal. No es legal sumar dos apuntadores, multiplicarlos o dividirlos, enmascararlos o agregarles un float o un double, o aún, excepto para void *, asignar un apuntador de un tipo a un apuntador de otro tipo sin una conversión forzosa de tipo.

5.5 Apuntadores a caracteres, y funciones

Una *constante de cadena*, escrita como

```
"Soy una cadena"
```

es un arreglo de caracteres. En la representación interna, el arreglo termina con el carácter nulo '\0', de tal manera que los programas puedan encontrar el fin. La longitud de almacenamiento es así uno más que el número de caracteres entre las comillas.

Posiblemente la más común ocurrencia de cadenas constantes se encuentra como argumentos a funciones, como en

```
printf("hola, mundo\n");
```

Cuando una cadena de caracteres como ésta aparece en un programa, el acceso a ella es a través de un apuntador a caracteres; printf recibe un apuntador al inicio del arreglo de caracteres. Esto es, se tiene acceso a una cadena constante por un apuntador a su primer elemento.

Las cadenas constantes no necesitan ser argumentos de funciones. Si pmessage se declara como

```
char *pmessage;
```

entonces la proposición

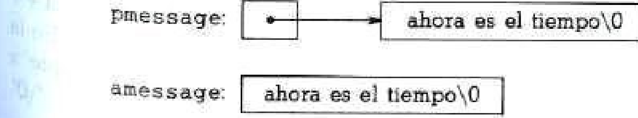
```
pmessage = "ya es el tiempo";
```

asigna a pmessage un apuntador al arreglo de caracteres. Esta *no* es la copia de una cadena; sólo concierne a apuntadores. El lenguaje C no proporciona ningún operador para procesar como unidad una cadena de caracteres.

Existe una importante diferencia entre estas definiciones:

```
char amessage[] = "ya es el tiempo";   /* arreglo */
char *pmessage = "ya es el tiempo";    /* apuntador */
```

amessage es un arreglo, suficientemente grande como para contener la secuencia de caracteres y el '\0' que lo inicializa. Se pueden modificar caracteres individuales dentro del arreglo, pero amessage siempre se referirá a la misma localidad de almacenamiento. Por otro lado, pmessage es un apuntador, inicializado para apuntar a una cadena constante; el apuntador puede modificarse posteriormente para que apunte a algún otro lado, pero el resultado es indefinido si trata de modificar el contenido de la cadena.



Ilustraremos más aspectos de los apuntadores y los arreglos, estudiando versiones de dos útiles funciones adaptadas de la biblioteca estándar. La primera función es **strcpy(s,t)**, que copia la cadena t a la cadena s. Sería agradable decir simplemente s=t, pero esto copia el apuntador, no los caracteres. Para copiar los caracteres se requiere de un ciclo. Primero está la versión con un arreglo:

```
/* strcpy: copia t hacia s; versión de subíndices */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t [i]) != '\0')
        i++;
}
```

En contraste, aquí está una versión de strcpy con apuntadores:

```
/* strcpy: copia t hacia s; versión 1 con apuntadores */
void strcpy(char *s, char *t)
{
    while ((*s = *t) = '\0') {
        s++;
        t++;
    }
}
```

Puesto que los argumentos se pasan por valor, strcpy puede utilizar los parámetros s y t en la forma que le parezca mejor. Aquí hay apuntadores convenientemente inicializados, que se desplazan a lo largo del arreglo un carácter a la vez, hasta que el '\0' con que termina t se ha copiado a s.

En la práctica, strcpy no se escribiría como se mostró anteriormente. Los programadores expertos de C preferirían

```
/* strcpy: copia t hacia s; versión 2 con apuntadores */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

Esto traslada el incremento de s y de t hacia dentro de la parte de prueba del ciclo. El valor de *t++ es el carácter al que apunta t antes de incrementarse; el ++ postfijo no modifica t sino hasta después de que se ha tomado el carácter. En la misma forma, el carácter se almacena en la posición anterior de s antes de que s se incremente. También este carácter es el valor contra el cual se compara '\0' para controlar el ciclo. El efecto real es que los caracteres se copian de t a s, hasta el '\0' final, incluyéndolo.

Como resumen final, observe que una comparación contra '\0' es redundante, puesto que la pregunta es simplemente si la expresión es cero. Así, la función podría escribirse correctamente como

```
/* strcpy: copia t hacia s; versión 3 con apuntadores */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Aunque esto puede parecer misterioso a primera vista, la conveniencia de esta notación es considerable, y debe dominarse el estilo, puesto que se encontrará frecuentemente en programas de C.

En la biblioteca estándar (<string.h>) strcpy, devuelve la cadena objetivo como el valor de la función.

La segunda rutina que examinaremos es **strcmp(s,t)**, que compara las cadenas de caracteres s y t, y regresa un valor negativo, cero o positivo si s es lexicográficamente menor que, igual a, o mayor que t. El valor se obtiene al restar los caracteres de la primera posición en que s y t no coinciden.

```
/* strcmp: regresa <0 si s<t, 0 si s==t, >0 si s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t [i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

La versión con apuntadores de strcmp:

```
/* strcmp: regresa <0 si s<t, 0 si s==t, >0 si s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Puesto que ++ y -- son operadores prefijos o postfijos, se presentan otras combinaciones de *, ++ y --, aunque con menos frecuencia. Por ejemplo,

```
*--p
```

disminuye p antes de traer el carácter al que apunta. En efecto, la pareja de expresiones

```
*p++ = val;     /* mete val en la pila */
val = *--p;     /* saca el tope de la pila y lo pone en val */
```

son expresiones idiomáticas estándar para meter y sacar algo de una pila; véase la sección 4.3.

El *header* <string.h> contiene declaraciones para las funciones que se mencionan en esta sección, además de una variedad de otras funciones para manipulación de cadenas en la biblioteca estándar.

Ejercicio 5-3. Escriba una versión con apuntadores de la función **strcat** que se muestra en el capítulo 2: **strcat(s,t)** copia la cadena **t** al final de **s**. □

Ejercicio 5-4. Escriba la función **strend(s,t)**, que regresa 1 si la cadena **t** se presenta al final de la cadena **s**, y cero si no es así. □

Ejercicio 5-5. Escriba versiones de las funciones de biblioteca **strncmp**, que operan con hasta los **n** primeros caracteres de sus argumentos de cadena. Por ejemplo, **strncpy(s,t,n)** copia hasta **n** caracteres de **t** hacia **s**. En el apéndice B se exponen descripciones más completas. □

Ejercicio 5-6. Reescriba los programas y ejercicios apropiados de los capítulos anteriores, empleando apuntadores en lugar de índices de arreglos. Buenos prospectos son **getline** (capítulos 1 y 4), **atoi**, **itoa**, y sus variantes (capítulos 2, 3 y 4), **reverse** (capítulo 3) y **strindex** y **getop** (capítulo 4). □

5.6 Arreglos de apuntadores; apuntadores a apuntadores

Puesto que en sí mismos los apuntadores son variables, pueden almacenarse en arreglos tal como otras variables. Ilustraremos esto al escribir un programa que ordena un conjunto de líneas de texto en orden alfabético, que es una versión restringida del programa **sort** de UNIX.

En el capítulo 3 se presentó una función de ordenamiento Shell que podía ordenar un arreglo de enteros, y en el capítulo 4 se mejoró con un *quicksort*. El mismo algoritmo funcionará, excepto que ahora se debe tratar con líneas de texto de diferentes longitudes, y que, a diferencia de los enteros, no se pueden comparar ni cambiar en una simple operación. Se necesita una representación de datos que maneje eficiente y convenientemente líneas de texto.

Aquí es donde entran los arreglos de apuntadores. Si las líneas que se van a ordenar se almacenan juntas en un gran arreglo de caracteres, entonces se puede tener acceso a cada línea por medio de un apuntador a su primer carácter. Por su lado, los apuntadores se pueden almacenar en un arreglo. Dos líneas se pueden comparar pasando sus apuntadores a **strcmp**. Cuando dos líneas desordenadas tienen que intercambiarse, se intercambian los apuntadores en el arreglo de apuntadores, no las líneas de texto.



Esto elimina el doble problema de un manejo complicado de almacenamiento y exceso de procesamiento que se produciría al mover las líneas.

El proceso de ordenamiento tiene tres pasos:

lee todas las líneas de entrada
ordénalas
imprímelas en orden

Como es usual, es mejor dividir el programa en funciones que coincidan con esta división natural, con la rutina principal controlando a las otras funciones. Abandonemos por un momento el paso de ordenamiento, y concentrémonos en las estructuras de datos y la entrada y salida.

La rutina de entrada tiene que reunir y guardar los caracteres de cada línea, y construir un arreglo de apuntadores hacia las líneas. También debe contar el número de líneas de entrada, puesto que esa información se requiere para el ordenamiento y la impresión. Debido a que la función de entrada sólo puede tratar con un número finito de líneas, puede regresar alguna cuenta de líneas ilegal, como −1, si se presenta demasiado texto a la entrada.

La rutina de salida sólo tiene que imprimir las líneas en el orden en que aparecen en el arreglo de apuntadores.

```
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000          /* máx # de líneas por ordenar */

char *lineptr[MAXLINES];       /* apuntadores a líneas de texto */

int readlines(char *lineptr[ ], int nlines);
void writelines(char *lineptr[ ], int nlines);

void qsort(char *lineptr[ ], int left, int right);

/* ordena líneas de entrada */
main( )
{
```

```
    int nlines;         /* número de líneas de entrada leídas */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: entrada demasiado grande para ordenarla\n");
        return 1;
    }
}
#define MAXLEN 1000     /* máx longitud de cualquier línea de entrada */
int getline(char *, int);
char *alloc(int);

/* readlines: lee líneas de entrada */
int readlines(char *lineptr[ ], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* elimina carácter nueva línea */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines:  escribe líneas de salida */
void writelines(char *lineptr[ ], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

La función getline se trató en la sección 1.9.

El principal nuevo elemento es la declaración para lineptr:

```
char *lineptr[MAXLINES]
```


que indica que lineptr es un arreglo de **MAXLINES** elementos, cada uno de los cuales es un apuntador a **char**. Esto es, **lineptr[i]** es un apuntador a carácter, y ***lineptr[i]** es el carácter al que apunta, el primer carácter de la i-ésima línea de texto almacenada.

Puesto que **lineptr** es por sí mismo el nombre de un arreglo, puede tratarse como un apuntador en la misma forma que en nuestros ejemplos anteriores, y writelines puede escribirse en su lugar como

```
/* writelines: escribe líneas de salida */
void writelines(char *lineptr[ ], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Inicialmente ***lineptr** apunta a la primera línea; cada incremento lo avanza al siguiente apuntador a línea mientras **nlines** se disminuye.

Teniendo la entrada y la salida bajo control, podemos proceder a ordenar. El *quicksort* del capítulo 4 necesita sólo cambios de poca importancia: las declaraciones deben modificarse, y la operación de comparación debe hacerse llamando a **strcmp**. El algoritmo permanece igual, lo que nos da cierta confianza de que aún trabajará.

```
/* qsort: ordena v[left]...v[right] en orden ascendente */
void qsort(char *v[ ], int left, int right)
{
    int i, last;
    void swap(char *v[ ], int i, int j);

    if (left >= right)      /* no hace nada si el arreglo contiene */
        return;             /* menos de dos elementos */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

De manera semejante, la rutina de intercambio sólo requiere modificaciones poco significativas:

```
/* swap:    intercambia v[i] y v[j] */
void swap(char *v[ ], int i, int j);
{
```

```
        char *temp;

        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
}
```

Puesto que cualquier elemento individual de v (alias lineptr) es un apuntador a carácter, temp también debe serlo, de modo que uno pueda copiarse al otro.

Ejercicio 5-7. Reescriba readlines para almacenar líneas en un arreglo proporcionado por main, en lugar de llamar a alloc para obtener espacio de almacenamiento. ¿Cuánto más rápido es el programa? □

5.7 Arreglos multidimensionales

El lenguaje C proporciona arreglos multidimensionales rectangulares, aunque en la práctica se usan menos que los arreglos de apuntadores. En esta sección mostraremos algunas de sus propiedades.

Considérese el problema de la conversión de fechas, de día del mes a día del año y viceversa. Por ejemplo, el 1 de marzo es el 60° día de un año que no es bisiesto, y el 61° día de uno que sí lo es. Definamos dos funciones para hacer la conversión: day_of_year convierte mes y día en el día del año, y month_day convierte el día del año en mes y día. Puesto que esta última función calcula dos valores, los argumentos de mes y día deben ser apuntadores:

```
    month_day(1988, 60, &m, &d)
```

hace m igual a 2 y d igual a 29 (29 de febrero).

Ambas funciones necesitan la misma información, una tabla de los números de días de cada mes ("treinta días tiene septiembre ..."). Puesto que el número de días por mes difiere para años bisiestos y no bisiestos, es más fácil separarlos en dos renglones de un arreglo bidimensional que siga la pista de lo que le pasa a febrero durante los cálculos. El arreglo y las funciones que realizan las transformaciones son como se muestra a continuación:

```
    static char daytab[2] [13] = {
        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    };

    /* day_of_year: obtiene día del año a partir de mes y año */
    int day_of_year(int year, int month, int day)
    {
        int i, leap;

        leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
        for (i = 1; i < month; i++)
            day += daytab[leap][i];
        return day;
    }

    /* month_day:  obtiene mes, y día a partir de día del año */
    void month_day(int year, int yearday, int *pmonth, int *pday)
    {
        int i, leap;

        leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
        for (i = 1; yearday > daytab[leap][i]; i++)
            yearday -= daytab[leap][i];
        *pmonth = i;
        *pday = yearday;
    }
```


Recuérdese que el valor aritmético de una expresión lógica, como la de leap, es cero (falso) o uno (verdadero), así que puede emplearse como índice del arreglo daytab.

El arreglo daytab tiene que ser externo tanto a day_of_year como a month_day, para que ambas puedan utilizarlo. Lo hicimos char para ilustrar un uso legítimo de char para almacenar enteros pequeños que no son caracteres.

daytab es el primer arreglo de caracteres de dos dimensiones con el que hemos tratado. En C, un arreglo de dos dimensiones es en realidad un arreglo unidimensional, cada uno de cuyos elementos es un arreglo. Por ello, los subíndices se escriben como

```
    daytab[i] [j]      /* [renglón] [columna] */
```

en lugar de

```
    daytab[i, j]       /* INCORRECTO */
```

Aparte de esta diferencia de notación, un arreglo de dos dimensiones puede tratarse en forma muy semejante a la de los otros lenguajes. Los elementos se almacenan por renglones, así que el índice de más a la derecha, o columna, varía más rápido cuando se tiene acceso a los elementos en orden de almacenamiento.

Un arreglo se inicializa con una lista de inicializadores entre llaves; cada renglón de un arreglo de dos dimensiones se inicializa con una sublista. El arreglo daytab se inicia con una columna de ceros, de modo que los números de mes puedan variar entre 1 y 12 en lugar de 0 a 11. Puesto que el espacio no es apremiante aquí, esto es más claro que ajustar los índices.

Si un arreglo de dos dimensiones se pasa a una función, la declaración de parámetros en la función debe incluir el número de columnas; el número de renglo-

nes es irrelevante, puesto que lo que se pasa es, como antes, un apuntador a un arreglo de renglones, donde cada renglón es un arreglo de 13 ints. Es este caso particular, es un apuntador a objetos que son arreglos de 13 ints. Entonces, si el arreglo **daytab** se pasara a la función f, la declaración de f sería

```
f(int daytab[2] [13]) { ... }
```

También podría ser

```
f(int daytab[ ] [13]) { ... }
```

porque el número de renglones es irrelevante, o podría ser

```
f(int (*daytab) [13]) { ... }
```

que indica que el parámetro es un apuntador a un arreglo de 13 enteros. Los paréntesis son necesarios, puesto que los corchetes [ ] tienen más alta precedencia que *. Sin paréntesis, la declaración

```
int *daytab [13]
```

es un arreglo de 13 apuntadores a entero. De modo más general, sólo la primera dimensión (subíndice) de un arreglo queda abierta; todas las otras deben especificarse.

En la sección 5.12 se discute más acerca de declaraciones complicadas.

Ejercicio 5-8. No existe detección de errores en **day_of_year** ni en **month_day**. Solucione ese defecto. □

5.8 Inicialización de arreglos de apuntadores

Considérese el problema de escribir una función **month_name(n)**, que regrese un apuntador a una cadena de caracteres que contengan el nombre del **n**-ésimo mes. Esta es una aplicación ideal para un arreglo static interno, **month_name** contiene un arreglo reservado de cadenas de caracteres, y regresa un apuntador a la cadena apropiada cuando se llama. Esta sección muestra como se inicializa ese arreglo de nombres.

La sintaxis es semejante a la de inicializaciones previas:

```
/* month_name:   regresa el nombre del n-ésimo mes */
char *month_name(int n)
{
    static char *name[ ] = {
        "Mes ilegal",
        "Enero", "Febrero", "Marzo",
        "Abril", "Mayo", "Junio",
        "Julio", "Agosto", "Septiembre",
        "Octubre", "Noviembre", "Diciembre"
    };
    return (n < 1 || n > 12) ? name[0] : name[n];
}
```


La declaración de **name**, que es un arreglo de apuntadores a caracteres, es la misma que la de **lineptr** en el ejemplo del ordenamiento. El inicializador es una lista de cadenas de caracteres; cada una se asigna a la posición correspondiente dentro del arreglo. Los caracteres de la i-ésima cadena se colocan en algún lugar, y en name[i] se almacena un apuntador a ellos. Puesto que el tamaño del arreglo name no está especificado, el compilador cuenta los inicializadores y completa el número correcto.

5.9 Apuntadores vs. arreglos multidimensionales

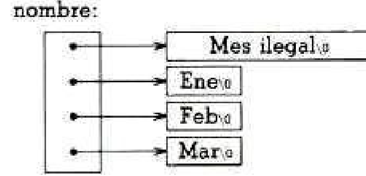
Los nuevos usuarios de C algunas veces se confunden con la diferencia entre un arreglo de dos dimensiones y uno de apuntadores, como **name** en el ejemplo anterior. Dadas las definiciones

```
int a[10] [20];
int *b[10];
```

entonces tanto **a[3] [4]** como **b[3] [4]** son referencias sintácticamente legítimas a un único **int**. Pero **a** es verdaderamente un arreglo de dos dimensiones: se le han asignado 200 localidades de tamaño de un int, y se emplea el cálculo convencional de subíndices rectangulares $20 \times renglón + columna$ para encontrar elemento a[*renglón*,*columna*]. Para b, sin embargo, la definición sólo asigna 10 apuntadores y no los inicializa; la inicialización debe realizarse en forma explícita, ya sea estáticamente o con código. Suponiendo que cada elemento de **b** apunta a un arreglo de veinte elementos, entonces existirán 200 ints reservados, más diez celdas para los apuntadores. La ventaja importante del arreglo de apuntadores es que los renglones del arreglo pueden ser de longitudes diferentes. Esto es, no es necesario que cada elemento de **b** apunte a un vector de veinte elementos; alguno puede apuntar a dos elementos, otro a cincuenta y algún otro a ninguno.

Aunque hemos basado esta discusión en términos de enteros, el uso más frecuente de arreglos de apuntadores es para almacenar cadenas de caracteres de longitudes diversas, como en la función **month_name**. Compare la declaración y la gráfica para un arreglo de apuntadores:

```
char *name[ ] = { "Mes ilegal", "Ene", "Feb", "Mar" };
```



con la de un arreglo bidimensional:

```
char aname[ ] [15] = { "Mes ilegal", "Ene", "Feb", "Mar" };
```

aname:

Mes ilegal\0	Ene\0	Feb\0	Mar\0
0	15	30	45

Ejercicio 5-9. Reescriba las rutinas **day_of_year** y **month_day** empleando apuntadores en lugar de índices. □

5.10 Argumentos en la línea de órdenes

Dentro de un medio ambiente que maneje C hay una forma de pasar argumentos en la línea de órdenes o de parámetros a un programa cuando empieza su ejecución. Cuando se llama a **main** se le invoca con dos argumentos. El primero (llamado por convención **argc**, por *argument count*) es el número de argumentos en la línea de órdenes con los que se invocó el programa; el segundo (**argv**, por *argument vector*) es un apuntador a un arreglo de cadenas de caracteres que contiene los argumentos, uno por cadena. Se acostumbra utilizar niveles múltiples de apuntadores para manipular esas cadenas de caracteres.

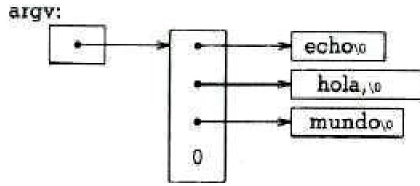
El ejemplo más sencillo es el programa **echo**, que despliega sus argumentos de la línea de órdenes en una línea, separados por blancos. Esto es, la orden

```
echo hola, mundo
```

imprime

```
hola, mundo
```

Por convención, **argv**[0] es el nombre con el que se invocó el programa, por lo que **argc** es por lo menos 1. Si **argc** es 1, entonce no hay argumentos en la línea después del nombre del programa. En el ejemplo anterior, **argc** es 3, y **argv**[0], **argv**[1] y **argv**[2] son "echo", "hola" y "mundo", respectivamente. El primer argumento optativo es **argv**[1] y el último es **argv**[**argc**—1]; además, el estándar requiere que **argv**[**argc**] sea un apuntador nulo.



La primera versión de **echo** trata a **argv** como un arreglo de apuntadores a caracteres:

```
#include <stdio.h>

/* eco de los argumentos de la línea de órdenes; 1a. versión */
main(int argc, char *argv[ ])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argv-1) ? " ": "");
    printf("\n");
    return 0;
}
```

Como **argv** es un apuntador a un arreglo de apuntadores, se pueden manipular al apuntador en lugar de indexar al arreglo. Esta siguiente variación se basa en incrementar **argv**, que es un apuntador a un apuntador a **char**, en tanto se disminuye **argc**:

```
#include <stdio.h>

/* eco de los argumentos de la línea de órdenes; 2a. versión */
main(int argc, char *argv[ ])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " :"");
    printf("\n");
    return 0;
}
```

Puesto que **argv** es un apuntador al inicio del arreglo de cadenas de argumentos, incrementarlo en 1 (++**argv**) lo hace apuntar hacia **argv**[1] en lugar de apuntar a **argv**[0]. Cada incremento sucesivo lo mueve al siguiente argumento; entonces ***argv** es el apuntador a ese argumento. Al mismo tiempo, **argc** disminuye; cuando llega a cero, no quedan argumentos por imprimir.

En forma alternativa, podemos escribir la proposición **printf** como

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Esto demuestra que el argumento de formato del **printf** también puede ser una expresión.

Como un segundo ejemplo, hagamos algunas mejoras al programa de la sección 4.1 que encuentra un patrón. Si se recuerda, se fijó el patrón de búsqueda en lo profundo del programa, un esquema que obviamente no es satisfactorio. Siguiendo la guía del programa **grep** de UNIX, cambiemos el programa de modo que el patrón que se debe encontrar se especifique por el primer argumento en la línea de órdenes.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:   imprime líneas que coinciden con el patrón del 1er. argumento */
main(int argc, char *argv[ ])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Uso: find patrón\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1] != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

La función strstr(s,t) de la biblioteca estándar regresa un apuntador a la primera ocurrencia de la cadena t dentro de la cadena s, o **NULL** si no existe. La cadena está declarada en <string.h>.

Ahora se puede extender el modelo para ilustrar construcciones adicionales de apuntadores. Suponga que deseamos permitir dos argumentos optativos. Uno indica "imprime todas las líneas *excepto* aquellas que coincidan con el patrón"; el segundo dice "precede cada línea impresa con su número de línea".

Una convención común para programas en C en sistemas UNIX es que un argumento que principia con un signo de menos introduce una bandera o parámetro optativo. Si seleccionamos **-x** (por "excepto") para indicar la inversión, y **-n** ("número") para solicitar la numeración de líneas, entonces la orden

find **-x -n** *patrón*

imprimirá cada línea que no coincida con el patrón, precedida por su número de línea.

Los argumentos para opciones deben ser permitidos en cualquier orden, y el resto del programa debe ser independiente del número de argumentos que estuvieran presentes. Además, es conveniente para los usuarios que los argumentos de las opciones puedan combinarse, como en

find **-nx** *patrón*

Aquí está el programa:


```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:   imprime líneas que coinciden con el patrón del 1er. argumento */
main(int argc, char *argv[ ])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: opción ilegal %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Uso: find -x -n patrón\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

argc se disminuye y **argv** se incrementa antes de cada argumento opcional. Al final del ciclo, si no hay errores, **argc** dice cuántos argumentos permanecen

sin procesar y **argv** apunta al primero de éstos. Así, **argc** debe ser 1 y ***argv** debe apuntar al patrón. Nótese que *++argv es un apuntador a un argumento tipo cadena, así que (*++argv)[0] es su primer carácter. (Una forma alternativa válida sería **++argv.) Debido a que [ ] tiene más prioridad que * y que ++, los paréntesis son necesarios; sin ellos, la expresión sería tomada como *++(argv[0]). En efecto, esto es lo que empleamos en el ciclo más interno, donde la tarea es proceder a lo largo de una cadena específica de argumentos. En el ciclo más interno, la expresión *++argv[0] incrementa el apuntador argv[0]

Es raro que se empleen expresiones con apuntadores más complicadas que éstas; en tal caso, será más intuitivo separarlas en dos o tres pasos.

Ejercicio 5-10. Escriba el programa **expr**, que evalúa una expresión polaca inversa de la línea de órdenes, donde cada operador u operando es un argumento por separado. Por ejemplo,

```
expr 2 3 4 + *
```

se evalúa como 2 × (3+4). □

Ejercicio 5-11. Modifique el programa entab y detab (escritos como ejercicios en el capítulo 1) para que acepten una lista de puntos de tabulación como argumentos. Utilice los tabuladores habituales si no hay argumentos. □

Ejercicio 5-12. Extienda **entab** y **detab** de modo que acepten la abreviatura

```
entab -m +n
```

que indica puntos de tabulación cada *n* columnas, iniciando en la columna *m*. Seleccione el comportamiento por omisión más conveniente (para el usuario). □

Ejercicio 5-13. Escriba el programa tail, que imprime las últimas *n* líneas de su entrada. Por omisión, *n* es 10, digamos, pero puede modificarse con un argumento optativo, de modo que

```
tail -n
```

imprime las últimas *n* líneas. El programa debe comportarse en forma racional sin importar cuán poco razonable sea la entrada o el valor de *n*. Escriba el programa de manera que haga el mejor uso de la memoria disponible; las líneas deben almacenarse como en el programa de ordenamiento de la sección 5.6, no en un arreglo de dos dimensiones de tamaño fijo. □

5.11 Apuntadores a funciones

En C, una función por sí sola no es una variable, pero es posible definir apuntadores a funciones, que pueden asignarse, ser colocados en arreglos, pasados a funciones, regresados por funciones y otras cosas más. Ilustraremos esto modificando el procedimiento de ordenamiento descrito anteriormente en este capítulo, de modo que si se da el argumento opcional -*n*, ordenará las líneas de entrada numéricamente en lugar de lexicográficamente.

Frecuentemente un ordenamiento consiste de tres partes —una comparación que determina el orden de cualquier par de objetos, un intercambio que invierte su orden, y un algoritmo de ordenamiento que realiza comparaciones e intercambios hasta que los objetos estén en orden. El algoritmo de ordenamiento es independiente de las operaciones de comparación e intercambio; así, al pasarle diferentes funciones de comparación e intercambio, se pueden obtener clasificaciones con diferentes criterios. Esta es la táctica que se sigue en nuestro nuevo método.

La comparación lexicográfica de dos líneas es realizada por **strcmp**, como antes; también requeriremos de una rutina **numcmp** que compare el valor numérico de dos líneas y regrese la misma clase de indicación que hace **strcmp**. Estas funciones se declaran antes de **main**, y a **qsort** se le pasa un apuntador a la función apropiada. Se ha hecho un procesamiento deficiente de los errores en los argumentos, con el fin de concentrarnos en los elementos principales.

```
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000          /* máx # de líneas a ordenar */
char *lineptr[MAXLINES];   /* apuntadores a líneas de texto */

int readlines(char *lineptr[ ], int nlines);
void writelines(char *lineptr[ ], int nlines);

void qsort(void *lineptr[ ], int left, int right,
                int (*comp) (void *, void *));
int numcmp(char *, char *);

/* ordena líneas de entrada */
main(int argc, char *argv[ ])
{
    int nlines;                /* número de líneas de entrada leídas */
    int numeric = 0;           /* 1 si es ordenamiento numérico */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
            (int (*) (void*, void*)) (numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
```

```
            printf ("entrada demasiado grande para ser ordenada\n");
            return 1;
        }
}
```

En la llamada a **qsort**, **strcmp** y **numcmp** son direcciones de funciones. Como se sabe que son funciones, el operador **&** no es necesario, en la misma forma que no es necesario antes del nombre de un arreglo.

Hemos escrito qsort de modo que pueda procesar cualquier tipo de dato, no sólo cadenas de caracteres. Como se indica por la función prototipo, qsort espera un arreglo de apuntadores, dos enteros y una función con dos argumentos de tipo apuntador. Para los argumentos apuntadores se emplea el tipo de apuntador genérico **void ***. Cualquier apuntador puede ser forzado a ser **void *** y regresado de nuevo sin pérdida de información, de modo que podemos llamar a **qsort** forzando los argumentos a **void***. El elaborado *cast* del argumento de la función fuerza los argumentos de la función de comparación. Esto generalmente no tendrá efecto sobre la representación real, pero asegura al compilador que todo esté bien.

```
/* qsort:   clasifica v[left]...v[right] en orden ascendente */
void qsort(void *v[ ], int left, int right,
           int (*comp) (void *, void *))
{
    int i, last;
    void swap(void *v[ ], int, int);

    if (left >= right)      /*no hace nada si el arreglo contiene */
        return;              /* menos de dos elementos */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp) (v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Las declaraciones deben estudiarse con cuidado. El cuarto parámetro de **qsort** es

```
int (*comp) (void *, void *)
```

que indica que **comp** es un apuntador a una función que tiene dos argumentos **void *** y regresa un **int**.

El uso de **comp** en la línea

```
if ((*comp) (v[i], v[left]) < 0)
```


es consistente con la declaración: **comp** es un apuntador a una función, ***comp** es la función, y

```
(*comp) (v[i], v[left])
```

es la llamada a ella. Los paréntesis son necesarios para que los componentes sean correctamente asociados; sin ellos,

```
int *comp(void *, void *)       /* INCORRECTO */
```

indica que **comp** es una función que regresa un apuntador a int, lo cual es muy diferente.

Ya hemos mostrado **strcmp**, que compara dos cadenas. Aquí está **numcmp**, que compara dos cadenas numéricamente, valor que se calcula llamando a **atof**:

```
#include <math.h>

/* numcmp:   compara s1 y s2 numéricamente */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

La función **swap**, que intercambia dos apuntadores, es idéntica a la que presentamos anteriormente en este capítulo, excepto en que las declaraciones se han cambiado a **void ***.

```
void swap(void *v[ ], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Puede agregarse una variedad de otras opciones al programa de ordenamiento; algunas se convierten en ejercicios interesantes.

Ejercicio 5-14. Modifique el programa de ordenamiento de modo que maneje una bandera **-r**, que indica ordenar en orden inverso (descendente). Asegúrese de que **-r**, trabaja con **-n**. □

Ejercicio 5-15. Agregue la opción -f para igualar las letras mayúsculas y minúsculas, de modo que no se haga distinción entre ellas durante el ordenamiento; por ejemplo, al comparar, **a** y **A** son iguales. □

Ejercicio 5-16. Agregue la opción -d ("orden de directorio"), que compara sólo letras, números y blancos. Asegúrese de que trabaja en conjunción con -f. □

Ejercicio 5-17. Agregue capacidad de manejo de campos, para que el ordenamiento se haga sobre campos de las líneas, cada campo ordenado de acuerdo con un conjunto independiente de opciones. (El índice de este libro fue ordenado con -df para las entradas y -n para los números de página.) □

5.12 Declaraciones complicadas

Al lenguaje C se le reprueba algunas veces por la sintaxis de sus declaraciones, particularmente las que involucran apuntadores a funciones. En la sintaxis hay un intento de hacer que coincidan las declaraciones con su uso; trabaja bien para casos simples, pero puede ser confusa para los difíciles, debido a que las declaraciones no pueden leerse de izquierda a derecha, y debido al exceso de uso de paréntesis. La diferencia entre

```
int *f( );          /* f: función que regresa un apuntador a int */
```

y

```
int (*pf)( );       /* pf: apuntador a una función que regresa un int */
```

ilustra el problema: * es un operador prefijo y tiene menor precedencia que ( ), de modo que los paréntesis son necesarios para obligar a una asociación apropiada.

Aunque en la práctica es extraño que aparezcan declaraciones verdaderamente complicadas, es importante saber cómo entenderlas y, si es necesario, cómo crearlas. Una buena forma de sintetizar declaraciones es en pequeños pasos con typedef, que se discute en la sección 6.7. Como una alternativa, en esta sección presentaremos un par de programas que convierten de C válido a una descripción verbal y viceversa. La descripción verbal se lee de izquierda a derecha.

La primera, dcl, es la más compleja. Convierte una declaración de C en una descripción hecha con palabras, como en estos ejemplos:

```
char **argv
    argv:   apuntador a un apuntador a char
int (*daytab) [13]
    daytab:  apuntador a un arreglo[13] de int
int *daytab[13]
    daytab:  arreglo[13] de apuntadores a int
void *comp( )
    comp:  función que regresa apuntador a void
```



```
void (*comp) ( )
    comp:     apuntador a una función que regresa void
char (*(*x( ))[ ])( )
    x:  función que regresa un apuntador a un arreglo[ ] de
    apuntadores a una función que regresa char
char (*(*x[3]) ( ))[5]
    x:  arreglo[3] de apuntadores a una función que regresa
    un apuntador a un arreglo[5] de char
```

dcl está basada en la gramática que especifica un declarador, que se define en forma precisa en el apéndice A, sección 8.5; ésta es una forma simplificada:

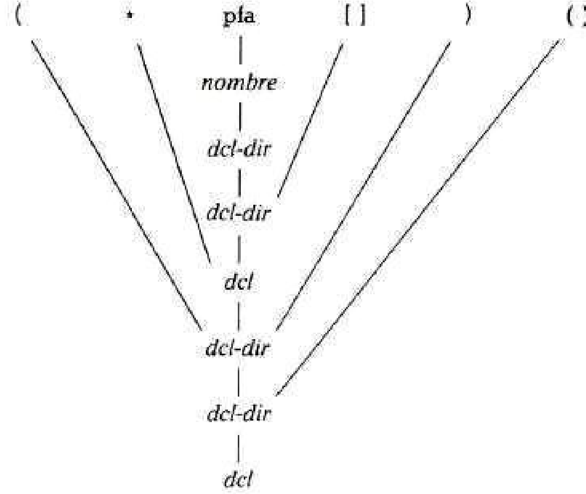
dcl:	**s optativos dcl-directa*
dcl-directa:	*nombre*
	(dcl)
	dcl-directo( )
	dcl-directa[tamaño optativo]

Con palabras, una *dcl* es una *dcl-directa*, tal vez precedido por * s. Una *dcl-directa* es un nombre, o una *dcl* entre paréntesis, o una *dcl-directa* seguida por paréntesis, o una *dcl-directa* seguida por corchetes con un tamaño optativo.

Esta gramática puede emplearse para reconocer declaraciones. Por ejemplo, considere este declarador:

```
(*pfa[ ]) ( )
```

pfa se identificará como un **nombre** y por ende como una *dcl-directa*. Entonces pfa[ ] es también una *dcl-directa*. Luego ***pfa[ ]** se reconoce como una *dcl*, de modo que **(*pfa[ ])** es una *dcl-directa*. Entonces **(*pfa[ ]) ( )** es una *dcl-directa* y por tanto una *dcl*. También podemos ilustrar el análisis con un árbol de estructura gramatical como éste (en donde *dcl-directa* se ha abreviado como *dcl-dir*):



El corazón del programa dcl es un par de funciones, dcl y dirdcl, que describen una declaración de acuerdo con esta gramática. Debido a que la gramática está definida recursivamente, las funciones se llaman recursivamente una a la otra, mientras reconocen piezas de una declaración; el programa se conoce como analizador sintáctico por descenso recursivo.

```
/* dcl: reconoce una declaración */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken( ) == '*'; )    /* cuenta *s */
        ns++;
    dirdcl( );
    while (ns-- > 0)
        strcat(out, "apuntador a");
}
/* dirdcl: reconoce un declarador directo */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {       /* ( dcl ) */
        dcl( );
        if (tokentype != ')')
            printf "error: falta )\n");
    } else if (tokentype == NAME) /* nombre de variable */
        strcpy(name, token);
    else
        printf("error: nombre o (dcl) esperado\n");
    while ((type=gettoken( )) == PARENS || type == BRACKETS)
        if (type == PARENS)
            strcat(out, "función que regresa");
        else {
            strcat(out, " arreglo");
            strcat(out, token);
            strcat(out, "de");
        }
}
```

Puesto que se intenta que el programa sea ilustrativo, no a prueba de balas, hay restricciones sobre dcl, que sólo puede manejar un tipo simple de datos como char o int. No maneja tipos de argumentos dentro de funciones, o calificadores como const. Los espacios en blanco inadecuados lo confunden. No se recupera mucho ante los errores, de modo que las declaraciones inválidas también lo confunden. Esas mejoras se dejan como ejercicios.

Aquí están las variables globales y la rutina principal:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;              /* tipo del último token */
char token[MAXTOKEN];       /* cadena del último token */
char name[MAXTOKEN];        /* nombre del identificador */
char datatype[MAXTOKEN];    /* tipo de dato = char, int, etc. */
char out[1000];             /* cadena de salida */

main( ) /* convierte una declaración a palabras */
{
    while (gettoken( ) != EOF) {       /* 1er. token en la línea */
        strcpy(datatype, token);  /* es el tipo de dato */
        out[0] = '\0';
        dcl( );           /* reconoce el resto de la línea */
        if (tokentype != '\n')
            printf("error de sintaxis\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

La función gettoken ignora blancos y tabuladores, y encuentra el siguiente token de la entrada; un "token" es un nombre, un par de paréntesis, un par de corchetes que tal vez incluyen un número, o cualquier otro carácter simple.

```
int gettoken(void)  /* regresa el siguiente token */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;

    while ((c = getch( )) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch( )) == ')' {
            strcpy(token, "( )");
```

```
            return tokentype = PARENS;
        } else {
            ungetch(c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch(c);
        return tokentype = NAME;
    } else
        return tokentype = c;
}
```

getch y ungetch se discutieron en el capítulo 4.

Es más fácil ir en la dirección inversa, especialmente si no nos preocupamos por la generación de paréntesis redundantes. El programa undcl convierte una descripción verbal como "x es una función que regresa un apuntador a un arreglo de apuntadores a funciones que regresan char", que se expresará como

```
x () * [] * () char
```

y se convertirá en

```
char (*(*x())[])()
```

La sintaxis abreviada de la entrada nos permite reutilizar a la función gettoken. La función undcl también emplea las mismas variables externas que dcl.

```
/* undcl: convierte una descripción verbal a declaración */
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf "entrada inválida en %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
```


Ejercicio 5-18. Haga que dcl se recupere de errores en la entrada. □

Ejercicio 5-19. Modifique undcl de modo que no agregue paréntesis redundantes a las declaraciones. □

Ejercicio 5-20. Extiende dcl para que maneje declaraciones con tipos de argumentos de funciones, calificadores como const, etcétera. □

CAPITULO 6: Estructuras

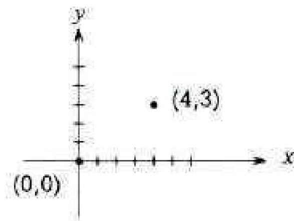
Una estructura es una colección de una o más variables, de tipos posiblemente diferentes, agrupadas bajo un solo nombre para manejo conveniente. (Las estructuras se conocen como "records" en algunos otros lenguajes, principalmente Pascal.) Las estructuras ayudan a organizar datos complicados, en particular dentro de programas grandes, debido a que permiten que a un grupo de variables relacionadas se les trate como una unidad en lugar de como entidades separadas.

Un ejemplo tradicional de estructura es el registro de una nómina: un empleado está descrito por un conjunto de atributos, como nombre, domicilio, número del seguro social, salario, etc. Algunos de estos atributos pueden, a su vez, ser estructuras: un nombre tiene varios componentes, como los tiene un domicilio y aún un salario. Otro ejemplo, más típico para C, procede de las gráficas: un punto es un par de coordenadas, un rectángulo es un par de puntos, y otros casos semejantes.

El principal cambio realizado por el estándar ANSI es la definición de la asignación de estructuras —las estructuras se pueden copiar y asignar, pasar a funciones y ser regresadas por funciones. Esto ha sido manejado por muchos compiladores durante varios años, pero las propiedades están ahora definidas en forma precisa. Las estructuras y los arreglos automáticos ahora también se pueden inicializar.

6.1 Conceptos básicos sobre estructuras

Definamos algunas estructuras propias para graficación. El objeto básico es un punto, del cual supondremos que tiene una coordenada x y una coordenada y, ambas enteras.



Los dos componentes pueden ser colocados en una estructura declarada así:

```
struct point {
      int x;
      int y;
};
```

La palabra reservada struct presenta la declaración de una estructura, que es una lista de declaraciones entre llaves. Un nombre optativo, llamado *rótulo de estructura*, puede seguir a la palabra struct (como aquí lo hace point). El rótulo da nombre a esta clase de estructura, y en adelante puede ser utilizado como una abreviatura para la parte de declaraciones entre llaves.

Las variables nombradas dentro de la estructura se llaman *miembros*. Un miembro de estructura o rótulo, y una variable ordinaria (esto es, no miembro) pueden tener el mismo nombre sin conflicto, puesto que siempre se pueden distinguir por el contexto. Además, en diferentes estructuras pueden encontrarse los mismos nombres de miembros, aunque por cuestiones de estilo se deberían de usar los mismos nombres sólo para objetos estrechamente relacionados.

Una declaración struct define un tipo. La llave derecha que termina la lista de miembros puede ser seguida por una lista de variables, como se hace para cualquier tipo básico. Esto es,

```
struct { ... } x, y, z;
```

es sintácticamente análogo a

```
int x, y, z;
```

en el sentido de que cada proposición declara a **x**, **y** y **z** como variables del tipo nombrado y causa que se les reserve espacio contiguo.

Una declaración de estructura que no está seguida por una lista de variables no reserva espacio de almacenamiento sino que simplemente describe una plantilla o la forma de una estructura. Sin embargo, si la declaración está rotulada, el rótulo se puede emplear posteriormente en definiciones de instancias de la estructura. Por ejemplo, dada la declaración anterior de **point**,

```
struct point pt;
```

define una variable pt que es una estructura de tipo struct point. Una estructura se puede inicializar al seguir su definición con una lista de inicializadores, cada uno una expresión constante, para los miembros:

```
struct point maxpt = { 320, 200 };
```

Una estructura automática también se puede inicializar por asignación o llamando a una función que regresa una estructura del tipo adecuado.

Se hace referencia a un miembro de una estructura en particular en una expresión con una construcción de la forma

nombre-estructura.miembro

El operador miembro de estructura "." conecta al nombre de la estructura con el nombre del miembro. Por ejemplo, para imprimir las coordenadas del punto pt,

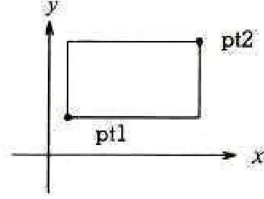
```
printf("%d,%d", pt.x, pt.y);
```

o para calcular la distancia del origen (0,0) a pt,

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Las estructuras pueden anidarse. Una representación de un rectángulo es como un par de puntos que denotan las esquinas diagonalmente opuestas:




```
struct rect {
      struct point pt1;
      struct point pt2;
};
```

La estructura **rect** contiene dos estructuras **point**. Si declaramos **screen** como

```
struct rect screen;
```

entonces

```
screen.pt1.x
```

se refiere a la coordenada *x* del miembro **pt1** de **screen**.

6.2 Estructuras y funciones

Las únicas operaciones legales sobre una estructura son copiarla o asignarla como unidad, tomar su dirección con &, y tener acceso a sus miembros. La copia y la asignación incluyen pasarlas como argumentos a funciones y también regresar valores de funciones. Las estructuras no se pueden comparar. Una estructura se puede inicializar con una lista de valores constantes de miembros; una estructura automática también se puede inicializar con una asignación.

Investiguemos las estructuras escribiendo algunas funciones para manipular puntos y rectángulos. Hay por lo menos tres acercamientos posibles: pasar separadamente los componentes, pasar una estructura completa o pasar un apuntador a ella. Cada uno tiene sus puntos buenos y malos.

La primera función, **makepoint**, toma dos enteros y regresa una estructura **point**:

```
/* makepoint:   crea un punto con las componentes x, y */
struct point makepoint(int x, int y)
{
      struct point temp;

      temp.x = x;
      temp.y = y;
      return temp;
}
```

Nótese que no hay conflicto entre el nombre del argumento y el miembro con el mismo nombre; incluso la reutilización de los nombres refuerza el vínculo.

makepoint ahora se puede usar para inicializar dinámicamente cualquier estructura, o para proporcionar los argumentos de la estructura a una función:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

El siguiente paso es un conjunto de funciones para hacer operaciones aritméticas sobre los puntos. Por ejemplo,

```
/* addpoint:   suma dos puntos */
struct point addpoint(struct point p1, struct point p2)
{
      p1.x += p2.x;
      p1.y += p2.y;
      return p1;
}
```

Aquí, tanto los argumentos como el valor de retorno son estructuras. Incrementamos los componentes en **p1** en lugar de utilizar explícitamente una variable temporal para hacer énfasis en que los parámetros de la estructura son pasados por valor como cualesquiera otros.

Como otro ejemplo, la función **ptinrect** prueba si un punto está dentro de un rectángulo, donde hemos adoptado la convención de que un rectángulo incluye sus lados izquierdo e inferior pero no sus lados superior y derecho:

```
/* ptinrect:   regresa 1 si p está en r, 0 si no lo está */
int ptinrect(struct point p, struct rect r)
{
      return p.x >= r.pt1.x && p.x < r.pt2.x
            && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Esto supone que el rectángulo está representado en una forma estándar en donde las coordenadas **pt1** son menores que las coordenadas **pt2**. La siguiente función regresa un rectángulo, garantizando que está en forma canónica:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect: pone en forma canónica las coordenadas de un rectángulo */
struct rect canonrect(struct rect r)
{
      struct rect temp;

      temp.pt1.x = min(r.pt1.x, r.pt2.x);
      temp.pt1.y = min(r.pt1.y, r.pt2.y);
      temp.pt2.x = max(r.pt1.x, r.pt2.x);
      temp.pt2.y = max(r.pt1.y, r.pt2.y);
      return temp;
}
```

Si una estructura grande va a ser pasada a una función, generalmente es más eficiente pasar un apuntador que copiar la estructura completa. Los apuntadores a estructuras son como los apuntadores a variables ordinarias. La declaración

```
struct point *pp;
```

dice que **pp** es un apuntador a una estructura de tipo **struct point**. Si **pp** apunta a una estructura **point**, ***pp** es la estructura, y **(*pp).x** y **(*pp).y** son los miembros. Para emplear **pp**, se podría escribir, por ejemplo,

```
struct point origin, *pp;

pp = &origin;
printf ("el origen es (%d,%d)\n", (*pp).x, (*pp).y;
```

Los paréntesis son necesarios en **(*pp).x** debido a que la precedencia del operador miembro de estructura . es mayor que la de *. La expresión ***pp.x** significa *(pp.x), lo cual es ilegal debido a que **x** no es un apuntador.

Los apuntadores a estructuras se usan con tanta frecuencia que se ha proporcionado una notación alternativa como abreviación. Si **p** es un apuntador a estructura, entonces

```
p->miembro de estructura
```

se refiere al miembro en particular. (El operador **->** es un signo menos seguido inmediatamente por >.) De esta manera podríamos haber escrito

```
printf("el origen es (%d,%d)\n", pp->x, pp->y);
```

Tanto . como -> se asocian de izquierda a derecha, de modo que si tenemos

```
struct rect r, *rp = r;
```

entonces estas cuatro expresiones son equivalentes:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Los operadores de estructuras . y ->, junto con ( ) para llamadas a funciones y [ ] para subíndices, están hasta arriba de la jerarquía de precedencias y se asocian estrechamente. Por ejemplo, dada la declaración

```
struct {
    int len;
    char *str;
} *p;
```

entonces

```
++p->len
```

incrementa a len, no a **p**, puesto que los paréntesis implícitos son **++(p->len)**. Los paréntesis se pueden emplear para alterar la asociación: **(++p)->len** incrementa a **p** antes de tener acceso a **len**, y **(++p)->len** incrementa a **p** después del acceso. (Este último conjunto de paréntesis es innecesario.)

De la misma manera, ***p->str** obtiene cualquier cosa a la que **str** apunte; ***p->str++** incrementa a **str** después de hacer el acceso a lo que apunta (exactamente como ***s++**); **(*p->str)++** incrementa cualquier cosa a la que **str** apunte; y ***p++->str** incrementa a **p** después de hacer el acceso a lo que **str** apunta.

6.3 Arreglos de estructuras

Considérese escribir un programa para contar las ocurrencias de cada palabra reservada de C. Se requiere de un arreglo de cadenas de caracteres para mantener

los nombres, y un arreglo de enteros para las cuentas. Una posibilidad es usar dos arreglos paralelos, **keyword** y **keyconunt**, como en

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Pero el hecho de que los arreglos sean paralelos sugiere una organización diferente, un arreglo de estructuras. Cada entrada de una palabra es una pareja:

```
char *word;
int count;
```

y hay un arreglo de parejas. La declaración de estructura

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

declara un tipo de estructura **key**, define un arreglo **keytab** de estructuras de ese tipo, y reserva espacio de almacenamiento para ellas. Cada elemento del arreglo es una estructura. Esto también se podría escribir como

```
struct key {
    char *word;
    int count;
};
struct key keytab[NKEYS];
```

Como la estructura **keytab** contiene un conjunto constante de nombres, es más fácil convertirla en una variable externa e inicializarla de una vez cuando se define. La inicialización de la estructura es análoga a otras anteriores —la definición es seguida por una lista de inicializadores entre llaves:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
```

```
        "volatile", 0,
        "while", 0
    };
```

Los inicializadores se listan en parejas correspondientes a los miembros de las estructuras. Podría ser más preciso encerrar los inicializadores para cada "renglón" o estructura entre llaves, como en

```
    { "auto", 0 },
    { "break", 0 },
    { "case", 0 },
    ...
```

pero las llaves internas no son necesarias cuando los inicializadores son variables simples o cadenas de caracteres, y cuando todos están presentes. Como es usual, el número de entradas en el arreglo **keytab** se calculará si los inicializadores están presentes y el [ ] se deja vacío.

El programa que cuenta palabras reservadas principia con la definición de **keytab**. La rutina principal lee de la entrada con llamadas repetidas a la función **getword**, que trae una palabra a la vez. Cada palabra se consulta en **keytab** con una versión de la función de búsqueda binaria que se escribió en el capítulo 3. La lista de palabras reservadas debe estar ordenada en forma ascendente en la tabla.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* cuenta palabras reservadas de C */
main( )
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count, keytab[n].word);
```



```
    return 0;
}

/* binsearch: encuentra una palabra en tab[0]. . .tab[n-1] */
int binsearch(char *word, struct key tab[ ], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Mostraremos la función **getword** en un momento; por ahora es suficiente decir que cada llamada a **getword** encuentra una palabra, que se copia dentro del arreglo referido como su primer argumento.

La cantidad **NKEYS** es el número de palabras en **keytab**. Aunque las podríamos contar manualmente, es mucho más fácil y seguro que lo haga la máquina, especialmente si la lista está sujeta a cambios. Una posibilidad sería terminar la lista de inicializadores con un apuntador nulo y luego hacer un ciclo a lo largo de **keytab** hasta que se encuentra el fin.

Pero esto es más de lo que se requiere, puesto que el tamaño del arreglo está determinado completamente al tiempo de compilación. El tamaño del arreglo es el tamaño de una entrada multiplicado por el número de entradas, así que el número de entradas es

size of **keytab** / *size of* struct key

C proporciona un operador unario a tiempo de compilación llamado **sizeof** que se puede emplear para calcular el tamaño de cualquier objeto. Las expresiones

sizeof *objeto*

y

sizeof (*nombre de tipo*)

dan un entero igual al tamaño en bytes del tipo u objeto especificado. (Estrictamente, **sizeof** produce un valor entero sin signo cuyo tipo, **size_t**, está definido en el header <**stddef.h**>.) Un objeto puede ser una variable, arreglo o estructura. Un nombre de tipo puede ser el nombre de un tipo básico como int o **double**, o un tipo derivado como una estructura o un apuntador.

En nuestro caso, el número de palabras es el tamaño del arreglo dividido entre el tamaño de un elemento. Este cálculo se utiliza en una proposición **#define** para fijar el valor de **NKEYS**:

```
#define   NKEYS   (sizeof keytab / sizeof(struct key))
```

Otra forma de escribir esto es dividir el tamaño del arreglo entre el tamaño de un elemento específico:

```
#define   NKEYS   (sizeof keytab / sizeof keytab[0])
```

Esto tiene la ventaja de que no necesita ser modificado si el tipo cambia.

Un **sizeof** no se puede utilizar en una línea **#if**, debido a que el preprocesador no analiza nombres de tipos. Pero la expresión del **#define** no es evaluada por el preprocesador, y aquí el código es legal.

Ahora la función **getword**. Hemos escrito una función **getword** más general de lo que se requiere para este programa, pero no es complicada. **getword** obtiene la siguiente "palabra" de la entrada, donde una palabra es cualquier cadena de letras y dígitos que principia con una letra, o un solo carácter que no sea espacio en blanco. El valor de la función es el primer carácter de la palabra, o **EOF** para fin de archivo, o el carácter en sí mismo cuando no es alfabético.

```
/* getword:   obtiene la siguiente palabra o carácter de la entrada */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch( )))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)){
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch( ))) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```


getword utiliza **getch** y **ungetch**, que se escribieron en el capítulo 4. Cuando la recolección de un símbolo alfanumérico se detiene, **getword** se ha colocado un carácter adelante. La llamada a **ungetch** regresa el carácter a la entrada para la siguiente llamada. **getword** también usa **isspace** para ignorar espacios en blanco, **isalpha** para identificar letras, e **isalnum** para identificar letras y dígitos; todas provienen del *header* estándar <**ctype.h**>.

Ejercicio 6-1. Nuestra versión de **getword** no maneja adecuadamente las subrayas, cadenas constantes, comentarios o líneas de control para el preprocesador. Escriba una versión mejorada. □

6.4 Apuntadores a estructuras

Para ilustrar algunas de las consideraciones involucradas con apuntadores y arreglos de estructuras, escribamos de nuevo el programa de conteo de palabras reservadas, esta vez utilizando apuntadores en lugar de subíndices.

La declaración externa de **keytab** no requiere de cambios, pero **main** y **binsearch** sí necesitan modificaciones.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* cuenta palabras reservadas de C; versión con apuntadores */
main( )
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}
```

```
/* binsearch:   encuentra una palabra en tab[0]. . .tab[n—1] */
struct key *binsearch(char *word, struct key *tab, int n)
{
    int cond;
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid— >word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

Aquí hay varias cosas que ameritan nota. Primero, la declaración de **binsearch** debe indicar que regresa un apuntador a **struct key** en lugar de un entero; esto se declara tanto en el prototipo de la función como en **binsearch**. Si **binsearch** encuentra la palabra, regresa un apuntador a ella; si no, regresa **NULL**.

Segundo, ahora se tiene acceso a los elementos de **keytab** por medio de apuntadores. Esto requiere de cambios significativos en binsearch.

Los inicializadores para **low** y **high** son ahora apuntadores al inicio y justo después del final de la tabla.

El cálculo del elemento intermedio ya no puede ser simplemente

```
mid = (low+high) / 2     /* INCORRECTO */
```

puesto que la suma de dos apuntadores es ilegal. Sin embargo, la resta es legítima, por lo que high-low es el número de elementos, y así

```
mid = low + (high-low) / 2
```

hace que **mid** apunte al elemento que está a la mitad entre low y high.

El cambio más importante es ajustar el algoritmo para estar seguros de que no genera un apuntador ilegal o intenta hacer acceso a un elemento fuera del arreglo. El problema es que &tab[—1] y &tab[n] están ambas fuera de los límites del arreglo **tab**. La primera es estrictamente ilegal, y es ilegal desreferenciar la segunda. Sin embargo, la definición del lenguaje garantiza que la aritmética de apuntadores que involucra el primer elemento después del fin de un arreglo (esto es, &tab[n] trabajará correctamente.

En **main** escribimos

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Si **p** es un apuntador a una estructura, la aritmética con **p** toma en cuenta el tamaño de la estructura, así que **p++** incrementa **p** con la cantidad correcta para obtener el siguiente elemento del arreglo de estructuras, y la prueba detiene el ciclo en el momento correcto.

Sin embargo, no hay que suponer que el tamaño de una estructura es la suma de los tamaños de sus miembros. Debido a requisitos de alineación para diferentes objetos, podría haber "huecos" no identificados dentro de una estructura. Así por ejemplo, si un **char** es de un byte y un **int** de cuatro bytes, la estructura

```
struct {
    char c;
    int i;
};
```

bien podría requerir ocho bytes, no cinco. El operador **sizeof** regresa el valor apropiado.

Finalmente, un comentario acerca del formato del programa: cuando una función regresa un tipo complicado como un apuntador a estructura, como en

```
struct key *binsearch(char *word, struct key *tab, int n)
```

el nombre de la función puede ser difícil de leer y de encontrar con un editor de texto. Por eso, algunas veces se emplea un estilo alternativo:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Esto es algo de gusto personal; seleccione la forma que prefiera y manténgala.

6.5 Estructuras autorreferenciadas

Supóngase que deseamos manejar el problema más general de contar las ocurrencias de *todas* las palabras en alguna entrada. Como la lista de palabras no se conoce por anticipado, no podemos ordenarlas convenientemente y utilizar una búsqueda binaria. No podemos hacer una búsqueda lineal para cada palabra que llega, para ver si ya se ha visto, puesto que el programa tomaría demasiado tiempo. (En forma más precisa, su tiempo de ejecución tiende a crecer en proporción cuadrática con el número de palabras de entrada.) ¿Cómo podemos organizar los datos para tratar eficientemente una lista de palabras arbitrarias?

Una solución es mantener siempre ordenado el conjunto de palabras que ya se han visto, colocando cada una en su posición correcta cuando llega. Esto,

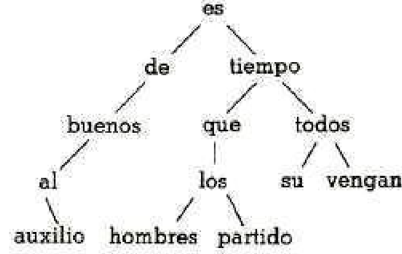
de cualquier manera, no se podría realizar recorriendo las palabras en un arreglo lineal —también tomado demasiado tiempo. En lugar de ello utilizaremos una estructura de datos llamada *árbol binario*.

El árbol contiene un "nodo" por cada palabra distinta; cada nodo contiene

```
un apuntador al texto de la palabra
una cuenta del número de ocurrencias
un apuntador al nodo hijo de la izquierda
un apuntador al nodo hijo de la derecha
```

Ningún nodo puede tener más de dos hijos; sólo puede tener cero o uno.

Los nodos se mantienen de tal manera que en cualquier nodo el subárbol izquierdo contiene sólo palabras que son lexicográficamente menores que la palabra que está en el nodo, y el subárbol de la derecha sólo contiene palabras que son mayores. Este es el árbol para la oración "Es tiempo de que todos los hombres buenos vengan al auxilio de su partido", como se construyó al insertar cada palabra tal como fue encontrada.



Para descubrir si una nueva palabra ya está en el árbol, inicie en la raiz y compárela con la que está almacenada en ese nodo. Si coincide, la pregunta se responde afirmativamente. Si la nueva palabra es menor que la palabra del árbol, continúe buscando en el nodo hijo de la izquierda o, de otra manera, en el nodo hijo de la derecha. Si ya no hay un hijo en la dirección requerida, la palabra nueva no está en el árbol, y de hecho la entrada vacía es el lugar apropiado para agregar la palabra nueva. Este proceso es recursivo, ya que la búsqueda desde cualquier nodo emplea una búsqueda desde uno de sus hijos. Por ello, unas rutinas recursivas para inserción e impresión serán lo más natural.

Regresando a la descripción de un nodo, se representa convenientemente como una estructura con cuatro componentes:

```
struct tnode {              /* el nodo del árbol: */
        char *word;                 /* apunta hacia el texto */
        int count;                  /* número de ocurrencias */
        struct tnode *left;         /* hijo a la izquierda */
        struct tnode *right;        /* hijo a la derecha */
};
```


Esta declaración recursiva de un nodo podría parecer riesgosa, pero es correcta. Es ilegal que una estructura contenga una instancia de sí misma, pero

```
struct tnode *left;
```

declara a left como un apuntador a tnode, no como un tnode en sí.

Ocasionalmente, se requiere de una variación de estructuras autorreferenciadas: dos estructuras que hagan referencia una a la otra. La forma de manejar esto es:

```
struct t {
    ...
    struct s *p;      /* p apunta a una s */
};
struct s {
    ...
    struct t *q;      /* q apunta a una t */
};
```

El código de todo el programa es sorprendentemente pequeño, dado un número de rutinas de soporte, como **getword**, que ya hemos descrito. La rutina principal lee palabras con **getword** y las instala en el árbol con **addtree**

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* conteo de frecuencia de palabras */
main( )
{

    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

La función **addtree** es recursiva. **main** presenta una palabra al nivel superior del árbol (la raíz). En cada etapa, la palabra se compara con la que ya está alma-

cenada en el nodo, y se filtra bajando hacia el subárbol izquierdo o derecho con una llamada recursiva a **addtree**. Finalmente la palabra coincidirá con algo que ya está en el árbol (en cuyo caso la cuenta se incrementa), o se encuentra un apuntador nulo, indicando que se debe crear un nodo y agregarlo al árbol. Si se crea un nuevo nodo, **addtree** regresa un apuntador a él, y lo instala en el nodo padre.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree:   agrega un nodo con w, en o bajo p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL){        /* llegó una nueva palabra */
        p = talloc( );      /* crea un nuevo nodo */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;         /* palabra repetida */
    else if (cond < 0)   /* menor que el contenido del subárbol izquierdo */
        p->left = addtree(p->left, w);
    else                  /* mayor que el contenido del subárbol derecho */
        p->right = addtree(p->right, w);
    return p;
}
```

El espacio de almacenamiento para el nuevo nodo se obtiene con la rutina **talloc**, la cual regresa un apuntador a un espacio libre adecuado para mantener un nodo del árbol, y la nueva palabra se copia a un lugar oculto con **strdup**. (Hablaremos de esas rutinas en un momento.) La cuenta se inicializa y los dos hijos se hacen nulos. Esta parte del código se ejecuta sólo para las hojas del árbol, cuando está siendo agregado un nuevo nodo. Hemos omitido (imprudencialmente) la revisión de errores en los valores regresados por **strdup** y **talloc**.

treeprint imprime el árbol en forma ordenada; para cada nodo escribe el subárbol izquierdo (todas las palabras menores que ésta), después la palabra en sí, y posteriormente el subárbol derecho (todas las palabras mayores). Si se siente inseguro sobre la forma en que trabaja la recursión, simule la operación de **treeprint** sobre el árbol mostrado anteriormente.

```
/* treeprint:     impresión del árbol p en orden */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
```



```
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Una nota práctica: si el árbol se "desbalancea" debido a que las palabras no llegan en orden aleatorio, el tiempo de ejecución puede aumentar demasiado. En el peor de los casos, si las palabras ya están en orden, este programa realiza una costosa simulación de búsqueda lineal. Existen generalizaciones del árbol binario que no padecen de este comportamiento del peor caso, pero no las describiremos aquí.

Antes de dejar este ejemplo, también es deseable una breve exposición sobre un problema relacionado con los asignadores de memoria. Es claramente deseable que sólo exista un asignador de almacenamiento en un programa, aun cuando asigne diferentes clases de objetos. Pero si un asignador va a procesar peticiones de, digamos, apuntadores a char y apuntadores a **struct tnodes**, surgen dos preguntas. Primera, ¿cómo cumple los requisitos de la mayor parte de las máquinas reales, de que los objetos de ciertos tipos deben satisfacer restricciones de alineación (por ejemplo, generalmente los enteros deben ser situados en localidades pares)? Segunda, ¿cuáles declaraciones pueden tratar con el hecho de que un asignador de memoria necesariamente debe regresar diferentes clases de apuntadores?

Los requisitos de alineación por lo general se pueden satisfacer fácilmente, al costo de algún espacio desperdiciado, asegurando que el asignador siempre regrese un apuntador que cumpla con todas las restricciones de alineación. El **alloc** del capítulo 5 no garantiza ninguna alineación en particular, de modo que emplearemos la función **malloc** de la biblioteca estándar, que sí lo hace. En el capítulo 8 se mostrará una forma de realizar **malloc**.

La pregunta acerca del tipo de declaración para una función como **malloc** es difícil para cualquier lenguaje que tome con seriedad la revisión de tipos. En C, el método apropiado es declarar que **malloc** regresa un apuntador a **void**, después forzar explícitamente con un *cast* al apuntador para hacerlo del tipo deseado, **malloc** y las rutinas relativas están declaradas en el *header* estándar <stdlib.h>. Así, talloc se puede escribir como

```
#include <stdlib.h>

/* talloc:      crea un tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

strdup simplemente copia la cadena dada por su argumento a un lugar seguro, obtenido por una llamada a **malloc**:

```
char *strdup(char *s)     /* crea un duplicado de s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1); /* +1 para '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

malloc regresa **NULL** si no hay espacio disponible; **strdup** pasa ese valor, dejando el manejo de error a su invocador.

El espacio obtenido al llamar a **malloc** puede liberarse para su reutilización llamando a **free**; véanse los capítulos 7 y 8.

Ejercicio 6-2. Escriba un programa que lea un programa en C e imprima en orden alfabético cada grupo de nombres de variable que sean idénticas en sus primeros 6 caracteres, pero diferentes en el resto. No cuente palabras dentro de cadenas ni comentarios. Haga que 6 sea un parámetro que pueda fijarse desde la línea de órdenes. □

Ejercicio 6-3. Escriba un programa de referencias cruzadas que imprima una lista de todas las palabras de un documento, y para cada palabra, una lista de los números de línea en los que aparece. Elimine palabras como "el", "y", etcétera. □

Ejercicio 6-4. Escriba un programa que imprima las distintas palabras de su entrada, ordenadas en forma descendente de acuerdo con su frecuencia de ocurrencia. Precede a cada palabra por su conteo. □

6.6 Búsqueda en tablas

En esta sección escribiremos los componentes de un paquete de búsqueda en tablas, para ilustrar más aspectos acerca de estructuras. Este código es típico de lo que podría encontrarse en las rutinas de manejo de tablas de símbolos de un macroprocesador o compilador. Por ejemplo, considere la proposición **#define**. Cuando se encuentra una línea como

```
#define  IN  1
```

el nombre **IN** y el texto de reemplazo 1 se almacenan en una tabla. Después, cuando el nombre **IN** aparece en una proposición como

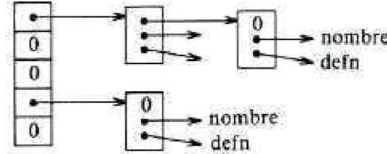
```
state = IN;
```

se debe reemplazar por 1.

Existen dos rutinas que manipulan los nombres y textos de reemplazo. **install(s,t)** registra el nombre **s** y el texto de reemplazo **t** en una tabla; **s** y **t** son

sólo cadenas de caracteres. **lookup(s)** busca **s** en la tabla y regresa un apuntador al lugar en donde fue encontrado, o **NULL** si no está.

El algoritmo es una búsqueda *hash* —el nombre que llega se convierte a un pequeño entero no negativo, que después se usa para indexar un arreglo de apuntadores. Un elemento del arreglo apunta al principio de una lista ligada de bloques que describen nombres que tienen ese valor de *hash*. El elemento es **NULL** si ningún nombre ha obtenido ese valor.



Un bloque de la lista es una estructura que contiene apuntadores al nombre, al texto de reemplazo y al siguiente bloque de la lista. Un siguiente apuntador nulo marca el final de la lista.

```
struct nlist {          /* entrada de la tabla: */
    struct nlist *next; /* siguiente entrada en la cadena */
    char *name;         /* nombre definido */
    char *defn;         /* texto de reemplazo */
};
```

El arreglo de apuntadores es sólo

```
#define HASHSIZE 101

static struct nlis *hashtab[HASHSIZE]; /* tabla de apuntadores*/
```

La función de *hash*, que se utiliza tanto en **lookup** como en **install**, agrega cada valor de carácter de la cadena a una combinación mezclada de los anteriores y regresa el módulo del residuo entre el tamaño del arreglo. Esta no es la mejor función de *hash* posible, pero es pequeña y efectiva.

```
/* hash:  forma un valor hash para la cadena s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

La aritmética sin signo asegura que el valor de *hash* no es negativo.

El proceso de *hash* produce un índice inicial en el arreglo hashtab; si la cadena se encontrara en algún lugar, será en la lista de bloques que empieza allí. La búsqueda se realiza por lookup. Si lookup encuentra que la entrada ya está presente, regresa un apuntador a ella; de otra manera, regresa **NULL**.

```
/* lookup:   busca s en hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;    /* se encontró */
    return NULL;          /* no se encontró */
}
```

El ciclo for que está en lookup es la expresión idiomática estándar para moverse sobre una lista ligada:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

install usa a lookup para determinar si el nombre que se va a instalar ya está presente; de ser así, la nueva definición toma el lugar de la anterior. De otra manera, se crea una nueva entrada. install regresa **NULL** si por cualquier razón no hay espacio para una nueva entrada.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: coloca (name, defn) dentro de hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) { /* no fue encontrado */
        np = (struct nlist *) malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else      /* ya está allí */
        free((void *) np->defn); /* libera la anterior defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```


Ejercicio 6-5. Escriba una función undef que borre un nombre y una definición de la tabla mantenida por lookup e install. □

Ejercicio 6-6. Haga una versión simple del procesador **#define** (esto es, sin argumentos) adecuada para usarse con programas en C, basada en las rutinas de esta sección. También podrá encontrar útiles getch y ungetch. □

6.7 Typedef

C proporciona una facilidad llamada **typedef** para crear nuevos tipos de datos. Por ejemplo, la declaración

```
typedef int Longitud;
```

hace del nombre **Longitud** un sinónimo de **int**. El tipo **Longitud** puede emplearse en declaraciones, *casts*, etc., exactamente de la misma manera en que lo podría ser int.

```
Longitud   len, maxlen;
Longitud   *lengths[ ];
```

De modo semejante, la declaración

```
typedef char *Cadena;
```

hace a **Cadena** un sinónimo para **char *** o apuntador a carácter, que después puede usarse en declaraciones y *casts*:

```
Cadena p, lineptr[MAXLINES], alloc(int);
int strcmp(Cadena, Cadena);
p = (Cadena) malloc(100);
```

Nótese que el tipo que se declara en un **typedef** aparece en la posición de un nombre de variable, no justo después de la palabra **typedef**. Sintácticamente, **typedef** es como las clases de almacenamiento **extern**, **static**, etc. Hemos empleado nombres con mayúscula para los **typedef**, para destacarlos.

Como un ejemplo más complicado, podríamos declarar mediante **typedef** los nodos del árbol mostrados anteriormente en este capítulo:

```
typedef struct tnode *Treeptr;

typedef struct tnode {   /* el nodo del árbol: */
    char *word;              /* apunta hacia el texto */
    int count;               /* número de ocurrencias */
    Treeptr left;            /* hijo izquierdo */
    Treeptr right;           /* hijo derecho */
} Treenode;
```

Esto crea dos nuevas palabras reservadas para tipos, llamados **Treenode** (una estructura) y **Treeptr** (un apuntador a la estructura). Entonces, la rutina **talloc** podría ser

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

Se debe destacar que una declaración **typedef** no crea un nuevo tipo en ningún sentido; simplemente agrega un nuevo nombre para algún tipo ya existente. Tampoco es alguna nueva semántica: las variables declaradas de esta manera tienen exactamente las mismas propiedades que las variables cuyas declaraciones se escriben explícitamente. En efecto, **typedef** es como **#define**, excepto que al ser interpretado por el compilador puede realizar substituciones textuales que están más allá de las capacidades del preprocesador. Por ejemplo,

```
typedef int (*AAF)(char *, char *);
```

crea el tipo **AAF**, de "apuntador a función (de dos argumentos **char ***) que regresa int", el cual se puede usar en contextos como

```
AAF strcmp, numcmp;
```

dentro del breve programa del capítulo 5.

Además de las razones puramente estéticas, hay dos razones principales para emplear **typedef**. La primera es parametrizar un programa contra los problemas de transportabilidad. Si se emplea **typedef** para tipos de datos que pueden ser dependientes de la máquina, cuando un programa se traslada, sólo los **typedef** requieren de cambios. Una situación común es usar nombres de **typedef** para varias cantidades enteras, y entonces hacer un conjunto apropiado de selecciones de **short**, **int** y **long** para cada máquina. Tipos como **size_t** y **ptrdiff_t** de la biblioteca estándar son ejemplos.

El segundo propósito de los **typedef** es proporcionar mejor documentación para un programa —un tipo llamado **Treeptr** puede ser más fácil de entender que uno declarado sólo como un apuntador a una estructura complicada.

6.8 Uniones

Una *unión* es una variable que puede contener (en momentos diferentes) objetos de diferentes tipos y tamaños, y el compilador hace el seguimiento del tamaño y requisitos de alineación. Las uniones proporcionan una forma de manipular diferentes clases de datos dentro de una sola área de almacenamiento, sin incluir en el programa ninguna información dependiente de la máquina. Son análogas a los *variant records* de Pascal.

Como un ejemplo, que podría ser encontrado en el manejador de la tabla de símbolos de un compilador, supóngase que una constante podría ser un int, un float, o un apuntador a carácter. El valor de una constante en particular debe ser guardado en una variable del tipo adecuado. No obstante, es conveniente para el manejador de tablas si el valor ocupa la misma cantidad de memoria y es guardado en el mismo lugar sin importar su tipo. Este es el propósito de una "unión" —una sola variable que puede legítimamente guardar uno de varios tipos. La sintaxis se basa en las estructuras:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

La variable u será suficientemente grande como para mantener al mayor de los tres tipos: el tamaño específico depende de la implantación. Cualquiera de estos tipos puede ser asignado a **u** y después empleado en expresiones, mientras que el uso sea consistente: el tipo recuperado debe ser el tipo que se almacenó más recientemente. Es responsabilidad del programador llevar el registro del tipo que está almacenado actualmente en una unión; si algo se almacena como un tipo y se recupera como otro, el resultado depende de la implantación.

Sintácticamente, se tiene acceso a los miembros de una unión con

nombre-unión.miembro

o

apuntador-unión->miembro

precisamente como a las estructuras. Si la variable **utype** se emplea para llevar el registro del tipo actualmente almacenado en u, entonces se podría ver el código como

```
if (utype == INT)
    printf("%d\n", u.ival);
else if (utype == FLOAT)
    printf("%f\n", u.fval);
else if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("dato incorrecto %d en utype\n", utype);
```

Las uniones pueden presentarse dentro de estructuras y arreglos, y viceversa. La notación para tener acceso a un miembro de una unión en una estructura (o viceversa) es idéntica a la de las estructuras anidadas. Por ejemplo, en el arreglo de estructuras definido por

```
struct {
    char *name;
    int flags;
```

```
        int utype;
        union {
            int ival;
            float fval;
            char *sval;
        } u;
    } symtab[NSYM];
```

al miembro ival se le refiere como

```
    symtab[i].u.ival
```

y al primer carácter de la cadena sval por cualquiera de

```
    *symtab[i].u.sval
    symtab[i].u.sval[0]
```

En efecto, una unión es una estructura en la cual todos los miembros tienen un desplazamiento de cero a partir de la base, la estructura es suficientemente grande para mantener al miembro "más ancho", y la alineación es la apropiada para todos los tipos de la unión. Están permitidas las mismas operaciones sobre las uniones como sobre las estructuras: asignación o copia como unidad, tomar la dirección, y hacer el acceso a un miembro.

Una unión sólo se puede inicializar con un valor del tipo de su pirimer miembro, así que la unión u descrita anteriormente sólo se puede inicializar con un valor entero.

El asignador de almacenamiento del capítulo 8 muestra cómo se puede usar una unión para obligar a que una variable sea alineada para una clase particular de límites de almacenamiento.

6.9 Campos de bits

Cuando el espacio de almacenamiento es escaso, puede ser necesario empaquetar varios objetos dentro de una sola palabra de máquina; un uso común es un conjunto de banderas de un bit en aplicaciones como tablas de símbolos para compiladores. Los formatos de datos impuestos externamente, como interfaces hacia dispostivos de hardware, frecuentemente requieren la capacidad de tomar partes de una palabra.

Imagínese un fragmento de un compilador que manipula una tabla de símbolos. Cada identificador dentro de un programa tiene cierta información asociada a él, por ejemplo, si es o no una palabra reservada, si es o no externa y/o estática y otros aspectos. La forma más compacta de codificar tal información es con un conjunto de banderas de un bit dentro de un **char** o **int**.

La forma usual en que esto se realiza es definiendo un conjunto de "máscaras" correspondientes a las posiciones relevantes de bits, como en


```
    #define KEYWORD 01
    #define EXTERNAL 02
    #define STATIC   04
```

o

```
    enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Los números deben ser potencias de dos. El acceso a los bits viene a ser cosa de "jugar" con los operadores de corrimiento, enmascaramiento y complemento, que se describieron en el capítulo 2.

Ciertas expresiones aparecen frecuentemente:

```
    flags |= EXTERNAL | STATIC;
```

enciende los bits **EXTERNAL** y **STATIC** en flags, en tanto que

```
    flags &= ~(EXTERNAL | STATIC);
```

los apaga, y

```
    if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

es verdadero si ambos bits están apagados.

Aunque estas expresiones se dominan fácilmente, como alternativa C ofrece la capacidad de definir y tener acceso a campos de una palabra más directamente que por medio de operadores lógicos de bits. Un *campo de bits*, o simplemente *campo*, es un conjunto de bits adyacentes dentro de una unidad de almacenamiento definida por la implantación, al que llamaremos "palabra". La sintaxis para la definición y acceso a campos está basada en estructuras. Por ejemplo, la anterior tabla de símbolos **#define** podría haber sido reemplazada por la definición de tres campos:

```
    struct {
        unsigned int is_keyword : 1;
        unsigned int is_extern  : 1;
        unsigned int is_static  : 1;
    } flags;
```

Esto define una variable llamada **flags**, que contiene tres campos de un bit. El número que sigue al carácter dos puntos representa el ancho del campo en bits. Los campos son declarados **unsigned int** para asegurar que sean cantidades sin signo.

Los campos individuales son referidos en la misma forma que para otros miembros de estructuras: **flags.is_keyword**, **flags.is_extern**, etc. Los campos se comportan como pequeños enteros y pueden participar en expresiones aritméticas, como lo hacen otros enteros. Así, el ejemplo previo pudo escribirse más naturalmente como

```
    flags.is_extern = flags.is_static = 1;
```

para encender los bits;

```
flags.is_extern = flags.is_static = 0;
```

para apagarlos; y

```
if (flag.is_extern == 0 && flags.is_static == 0)
...
```

para probarlos.

Casi todo acerca de los campos es dependiente de la implantación. El que un campo pueda traslapar al límite de una palabra se define por la implantación. Los campos no necesitan tener nombre; los campos sin nombre (dos puntos y su amplitud solamente) se emplean para llenar espacios. El ancho especial 0 puede emplearse para obligar a la alineación al siguiente límite de palabra.

Los campos se asignan de izquierda a derecha en algunas máquinas y de derecha a izquierda en otras. Esto significa que aunque los campos son útiles para el mantenimiento de estructuras de datos definidas internamente, la pregunta de qué punta viene primero tiene que considerarse cuidadosamente cuando se seleccionan datos definidos externamente; los programas que dependen de tales cosas no son transportables. Los campos sólo se pueden declarar como enteros; por transportabilidad, se debe especificar explícitamente **signed** o **unsigned**. No son arreglos y no tienen direcciones, de modo que el operador **&** no puede aplicarse a ellos.

CAPITULO 7: Entrada y salida

Las operaciones de entrada y salida no son en sí parte del lenguaje C, por lo que hasta ahora no las hemos destacado. Sin embargo, los programas interactúan con su medio ambiente en formas mucho más complicadas de las que hemos mostrado antes. En este capítulo describiremos la biblioteca estándar, un conjunto de funciones que proporcionan entrada y salida, manipulación de cadenas, manejo de memoria, rutinas matemáticas y una variedad de otros servicios para programas en C, aunque haremos hincapié en la entrada y salida.

El estándar ANSI define de manera precisa estas funciones de biblioteca, de modo que pueden existir en forma compatible en cualquier sistema en donde exista C. Los programas que restringen su interacción con el sistema a las facilidades provistas por la biblioteca estándar pueden ser llevados de un sistema a otro sin cambios.

Las propiedades de las funciones de biblioteca están especificadas en más de una docena de *headers*; ya hemos visto algunos, incluyendo <stdio.h>, <string.h> y <ctype.h>. No presentaremos aquí la totalidad de la biblioteca, puesto que estamos más interesados en escribir programas en C que los usan. La biblioteca se describe en detalle en el apéndice B.

7.1 Entrada y salida estándar

Como señalamos en el capítulo 1, la biblioteca consiste en un modelo simple de entrada y salida de texto. Un flujo de texto consiste en una secuencia de líneas, cada una de las cuales termina con un carácter nueva línea. Si el sistema no opera de ese modo, la biblioteca hace lo que sea necesario para simular que así funciona. Por ejemplo, la biblioteca podría convertir el regreso de carro y avance de línea a una nueva línea en la entrada y de nuevo en la salida.

El mecanismo de entrada más simple es leer un carácter a la vez de la *entrada estándar*, normalmente el teclado, con **getchar**:

```
int getchar(void)
```

getchar regresa el siguiente carácter de la entrada cada vez que se invoca, o **EOF** cuando encuentra fin de archivo. La constante simbólica **EOF** está definida en <stdio.h>. El valor es típicamente −1, pero las pruebas se deben escribir en función de **EOF**, de modo que sean independientes del valor específico.

En muchos medios ambientes, un archivo puede tomar el lugar del teclado empleando la convención < para redireccionamiento de entrada: si un programa prog usa **getchar**, entonces la línea de órdenes

```
prog <archent
```

provoca que **prog** lea caracteres de **archent**. El cambio de la entrada se realiza de tal manera que **prog** mismo es ajeno al cambio; en particular, la cadena "<archent" no está incluida entre los argumentos de la línea de órdenes en **argv**. El cambio de la entrada es también invisible si la entrada viene de otro programa vía un mecanismo de interconexión (*pipe*): en algunos sistemas, la línea de órdenes

```
otroprog | prog
```

ejecuta tanto al programa **otroprog** como a **prog**, e interconecta la salida estándar de **otroprog** con la entrada estándar para **prog**.

La función

```
int putchar(int)
```

se emplea para salida: **putchar(c)** coloca el carácter **c** en la *salida estándar*, que por omisión es la pantalla. **putchar** regresa el carácter escrito, o **EOF** si ocurre algún error. De nuevo, la salida puede ser dirigida hacia algún archivo con >*nombrearch*: si **prog** utiliza **putchar**,

```
prog >archsal
```

escribirá la salida estándar hacia **archsal**. Si se permite la interconexión,

```
prog | otroprog
```

deja la salida estándar de **prog** en la entrada estándar de **otroprog**.

La salida producida por **printf** también encuentra su camino hacia la salida estándar. Las llamadas a **putchar** y a **printf** pueden estar traslapadas —la salida aparece en el orden en que se hicieron las llamadas.

Cada archivo fuente que se refiera a una función de biblioteca de entrada/salida debe contener la línea

```
#include <stdio.h>
```

antes de la primera referencia. Cuando un nombre se delimita por < y > se realiza una búsqueda del *header* en algunos lugares estándar (por ejemplo, en los sistemas UNIX, típicamente en el directorio **/usr/include**).

Muchos programas leen sólo un flujo de entrada y escriben sólo un flujo de salida; para tales programas la entrada y salida con **getchar**, **putchar** y **printf**, puede ser totalmente adecuada y en realidad es suficiente para comenzar. Esto es particularmente cierto si se emplea la redirección para conectar la salida de un programa a la entrada de otro. Por ejemplo, considérese el programa **lower**, que convierte su entrada a minúsculas:


```
#include <stdio.h>
#include <ctype.h>

main( )   /* lower:   convierte la entrada a minúsculas */
{
    int c;

    while ((c = getchar( )) != EOF)
        putchar(tolower(c));
    return 0;
}
```

La función **tolower** está definida en <ctype.h>; convierte una letra mayúscula a minúscula, y regresa los otros caracteres intactos. Como mencionamos antes, las "funciones" como **getchar** y **putchar** en <stdio.h> y **tolower** en <ctype.h> son a menudo macros, evitándose así la sobrecarga de una llamada a función por cada carácter. En la sección 8.5 se mostrará cómo se hace esto. Sin importar cómo sean las funciones de <ctype.h> en una máquina dada, los programas que las emplean están aislados del juego de caracteres.
de caracteres.

Ejercicio 7-1. Escriba un programa que convierta mayúsculas a minúsculas o viceversa, dependiendo del nombre con que se invoque, dado en **argv[0]**. □

7.2 Salida con formato—printf

La función de salida **printf** traduce valores internos a caracteres. Ya hemos empleado informalmente **printf** en los capítulos anteriores. La descripción de aquí cubre los usos más típicos, pero no está completa; para la definición completa, véase el apéndice B.

```
int printf(char *format, arg1, arg2, ...)
```

printf convierte, da formato e imprime sus argumentos en la salida estándar bajo el control de **format**. Regresa el número de caracteres impresos.

La cadena de formato contiene dos tipos de objetos: caracteres ordinarios, que son copiados al flujo de salida, y especificaciones de conversión, cada uno de los cuales causa la conversión e impresión de los siguientes argumentos sucesivos de **printf**. Cada especificación de conversión comienza con un % y termina con un carácter de conversión. Entre el % y el carácter de conversión pueden estar, en orden:

- Un signo menos, que especifica el ajuste a la izquierda del argumento convertido.
- Un número que especifica el ancho mínimo de campo. El argumento convertido será impreso dentro de un campo de al menos este ancho. Si es necesario será llenado de blancos a la izquierda (o a la derecha, si se requiere ajuste a la izquierda) para completar la amplitud del campo.
- Un punto, que separa el ancho de campo de la precisión.
- Un número, la precisión, que especifica el número máximo de caracteres de una cadena que serán impresos, o el número de dígitos después del punto decimal de un valor de punto flotante, o el número mínimo de dígitos para un entero.
- Una h si el entero será impreso como un **short**, o una l (letra ele) si será como un long.

Los caracteres de conversión se muestran en la tabla 7-1. Si el carácter después del % no es una especificación de conversión, el comportamiento no está definido.

TABLA 7-1. CONVERSIONES BASICAS DE PRINTF

CARACTER	TIPO DE ARGUMENTO: IMPRESO COMO
d, i	int; número decimal.
o	int; número octal sin signo (sin cero inicial).
x, X	int; número hexadecimal sin signo (con un 0x o 0X inicial, usando **abcdef** o **ABCDEF** para 10, ... 15.
u	int; número decimal sin signo.
c	int; carácter sencillo.
s	char *; imprime caracteres de una cadena hasta un '\0' o el número de caracteres dado por la precisión.
f	double; [−]*m.dddddd*, en donde el número de *ds* está dado por la precisión (predeterminado a 6).
e, E	double; [−]*m.dddddd*e±*xx* o [−]*m.dddddd*E±*xx*, en donde el número de *ds* está dado por la precisión (predeterminado a 6).
g, G	double; usa %e o %E si el exponente es menor que −4 o mayor o igual a la precisión; de otra forma usa %f. Los ceros o el punto al final no se imprimen.
p	void *; apuntador (representación dependiente de la instalación).
%	no es convertido en ningún argumento; imprime un %.

Una amplitud o precisión se puede especificar por *, en cuyo caso el valor se calcula convirtiendo el siguiente argumento (que debe ser **int**). Por ejemplo, para imprimir al menos **max** caracteres de una cadena s,

```
printf("%.*s", max, s);
```


La mayoría de las conversiones de formato se han ilustrado en capítulos anteriores. Una excepción es la precisión relacionada con las cadenas. La siguiente tabla muestra el efecto de una variedad de especificaciones al imprimir "hola, mundo" (11 caracteres). Hemos colocado el carácter dos puntos alrededor de cada campo para que se pueda apreciar su extensión.

```
:%s:          :hola, mundo:
:%10s:        :hola, mundo:
:%.10s:       :hola, mund:
%-10s:        :hola, mundo:
:%.15s:       :hola, mundo:
:%-15s:       :hola, mundo    :
:%15.10s:     :     hola, mund:
:%-15.10s:    :hola, mund     :
```

Una advertencia: **printf** emplea su primer argumento para decidir cuántos argumentos le siguen y cuáles son sus tipos. **printf** se confundirá y se obtendrán resultados erróneos si no hay suficientes argumentos o si tienen tipos incorrectos. También debe advertir la diferencia entre estas dos llamadas:

```
printf(s);          /* FALLA si s contiene % */
printf("%s", s);    /* SEGURO */
```

La función **sprintf** realiza las mismas conversiones que printf, pero almacena la salida en de una cadena:

```
int sprintf(char *cadena, char *format, arg1, arg2, ...)
```

sprintf da formato a los argumentos que están en arg_1, arg_2, etc., de acuerdo con **format** como antes, pero coloca el resultado en cadena en vez de en la salida estándar; cadena debe ser suficientemente grande como para recibir el resultado.

Ejercicio 7-2. Escriba un programa que imprima una entrada arbitraria en forma sensata. Como mínimo, deberá imprimir caracteres no gráficos en octal o hexadecimal de acuerdo con la costumbre local, y separar líneas largas de texto. □

7.3 Listas de argumentos de longitud variable

Esta sección contiene la realización de una versión mínima de **printf**, para mostrar cómo escribir una función que procese una lista de argumentos de longitud variable en una forma transportable. Puesto que estamos interesados principalmente en el procesamiento de argumentos, **minprintf** procesará la cadena de formato y los argumentos, pero llamará al **printf** real para hacer las conversiones de formato.

La declaración correcta para **printf** es

```
int printf(char *fmt, ...)
```

donde la declaración ... significa que el número y tipo de esos argumentos puede variar. La declaración ... sólo puede aparecer al final de la lista de argumentos. Nuestra minprintf se declara como

```
void minprintf(char *fmt, ...)
```

ya que no regresará la cuenta de caracteres que regresa printf.

El truco está en cómo minprintf recorre la lista de argumentos cuando la lista ni siquiera tiene un nombre. El *header* estándar **<stdarg.h>** contiene un conjunto de macrodefiniciones que definen cómo avanzar sobre una lista de argumentos. La realización de este *header* variará de una máquina a otra, pero la interfaz que presenta es uniforme.

El tipo **va_list** se emplea para declarar una variable que se referirá a cada argumento en su momento; en **minprintf**, esta variable se llama **ap**, por "argument pointer" (apuntador a argumento). La macro **va_start** inicializa **ap** para apuntar al primer argumento sin nombre. Debe llamarse una vez antes de usar **ap**. Al menos debe haber un argumento con nombre; el último argumento con nombre es empleado por **va_start** para iniciar.

Cada llamada de **va_arg** regresa un argumento y avanza **ap** al siguiente; **va_arg** emplea un nombre de tipo para determinar qué tipo regresar y cuán grande será el avance. Finalmente, **va_end** realiza las labores de limpieza y arreglo que sean necesarias. Debe invocarse antes que la función regrese.

Estas propiedades forman la base de nuestro **printf** simplificado:

```
#include <stdarg.h>

/* minprintf:   printf mínima con lista variable de argumentos */
void minprintf(char *fmt, ...)
{
    va_list ap;   /* apunta a cada arg sin nombre en orden */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt);   /* hace que ap apunte al 1er. arg sin nombre */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap);   /* limpia cuando todo está hecho */
}
```


Ejercicio 7-3. Aumente **minprintf** para que maneje otras de las características de printf. □

7.4 Entrada con formato—scanf

La función **scanf** es la entrada análoga de **printf**, y proporciona muchas de las mismas facilidades de conversión en la dirección opuesta.

```
int scanf(char *format, ...)
```

scanf lee caracteres de la entrada estándar, los interpreta de acuerdo con las especificaciones que están en **format**, y almacena los resultados a través de los argumentos restantes. El argumento de formato se describe abajo; los otros argumentos, *cada uno de los cuales debe ser un apuntador*, indican dónde deberá almacenarse la entrada correspondientemente convertida. Como con **printf**, esta sección es un resumen de las posibilidades más útiles, no una lista exhaustiva.

scanf se detiene cuando termina con su cadena de formato, o cuando alguna entrada no coincide con la especificación de control. Regresa como su valor el número de ítems de entrada que coinciden con éxito. Esto se puede emplear para decidir cuántos ítems se encontraron. Al final del archivo, regresa EOF; nótese que esto es diferente de 0, que significa que el siguiente carácter de entrada no coincide con la primera especificación en la cadena de formato. La siguiente llamada a **scanf** continúa la búsqueda inmediatamente después del último carácter que ya fue convertido.

Existe también una función sscanf que lee de una cadena y no de la entrada estándar:

int sscanf(char *cadena, char *format, arg_1, arg_2, ...)

Rastrea la cadena de acuerdo con el formato en **format**, y almacena el valor resultante a través de arg_1, arg_2, etc. Estos argumentos deben ser apuntadores.

La cadena de formato generalmente contiene especificaciones de conversión, las cuales son empleadas para controlar la conversión de entrada. La cadena de formato puede contener:

- Blancos o tabuladores, los cuales son ignorados.
- Caracteres ordinarios (no %), que se espera coincidan con el siguiente carácter que no sea espacio en blanco del flujo de entrada.
- Especificaciones de conversión, consistentes en el carácter %, un carácter optativo de supresión de asignación *, un número optativo que especifica el ancho máximo de campo, una h, l, o L optativa que indica la amplitud del objetivo, y un carácter de conversión.

La especificación de conversión dirige la conversión del siguiente campo de entrada. Normalmente el resultado se coloca en la variable apuntada por el argumento correspondiente. Si se indica la supresión de asignación con el carácter *, sin embargo, el campo de entrada es ignorado y no se realiza asignación alguna. Un campo de entrada está definido como una cadena de caracteres que no son espacio en blanco; se extiende hasta el siguiente espacio en blanco o hasta que el ancho de campo se agote, si está especificado. Esto implica que **scanf** leerá entre varias líneas para encontrar su entrada, ya que las nuevas líneas son espacios en blanco. (Los caracteres de espacio en blanco son tabulador, nueva línea, retorno de carro, tabulador vertical y avance de hoja.)

El carácter de conversión indica la interpretación del campo de entrada. El argumento correspondiente debe ser un apuntador, como es requerido por la semántica de las llamadas por valor de C. Los caracteres de conversión se muestran en la tabla 7-2.

TABLA 7-2. CONVERSIONES BASICAS DE SCANF

CARACTER	DATO DE ENTRADA: TIPO DE ARGUMENTO:
d,	entero decimal; int *.
i	entero; int *. El entero puede estar en octal (iniciado con 0) o hexadecimal (iniciado con 0x o 0X).
o	entero octal (con o sin cero inicial); int *.
u	entero decimal sin signo; unsigned int *.
x	entero hexadecimal (iniciado o no con 0x o 0X); int *.
c	caracteres; char *. Los siguientes caracteres de entrada (por omisión 1) son colocados en el sitio indicado. El salto normal sobre los espacios en blanco es suprimido; para leer el siguiente espacio no blanco, use %1s.
s	cadena de caracteres (no entrecomillada); char *, apunta a un arreglo de caracteres suficientemente grande para la cadena y una terminación '\0' que será agregada.
e, f, g	número de punto flotante con signo, punto decimal y exponente optativos; float *.
%	% literal; no se hace asignación alguna.

Los caracteres de conversión d, i, o, u, x pueden ser precedidos por **h** para indicar que en la lista de argumentos aparece un apuntador a **short** en lugar de a int, o por l (letra ele) para indicar que aparece un apuntador a **long** en la lista de argumentos. En forma semejante, los caracteres de conversión e, f, g pueden ser precedidos por l para indicar que hay un apuntador a **double** en lugar de a float en la lista de argumentos.

Como un primer ejemplo, la rudimentaria calculadora del capítulo 4 se puede escribir con **scanf** para hacer la conversión de entrada:

```
#include    <stdio.h>

main( )    /* calculadora rudimentaria */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Suponga que deseamos leer líneas de entrada que contienen fechas de la forma

```
25 Dic 1988
```

La proposición **scanf** es

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

No se emplea & con **monthname**, ya que un nombre de arreglo es un apuntador.

Pueden aparecer caracteres literales en la cadena de formato de **scanf**, y deben coincidir con los mismos caracteres de la entrada. De modo que podemos leer fechas de la forma mm/dd/yy con esta proposición **scanf**:

```
int day, month, year;

scanf("%d/%d/%d", &month, &day, &year);
```

scanf ignora los blancos y los tabuladores que estén en su cadena de formato. Además, salta sobre los espacios en blanco (blancos, tabuladores, nuevas líneas, etc.) mientras busca los valores de entrada. Para leer de entradas cuyo formato no está fijo, a menudo es mejor leer una línea a la vez, y después separarla con **sscanf**. Por ejemplo, suponga que deseamos leer líneas que pueden contener fechas en cualquiera de las formas anteriores. Entonces podemos escribir

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("válido: %s\n", line);      /* forma 25 Dic 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("válido: %s\n", line);      /* forma mm/dd/yy */
    else
        printf("inválido: %s\n", line);    /* forma inválida */
}
```

Las llamadas a **scanf** pueden estar mezcladas con llamadas a otras funciones de entrada. La siguiente llamada a cualquier función de entrada iniciará leyendo el primer carácter no leído por scanf.

Una advertencia final: los argumentos de **scanf** y **sscanf** *deben* ser apuntadores. El error más común es escribir

```
scanf("%d", n);
```

en lugar de

```
scanf("%d", &n);
```

Este error generalmente no se detecta en tiempo de compilación.

Ejercicio 7-4. Escriba una versión privada de **scanf** análoga a **minprintf** de la sección anterior. □

Ejercicio 7-5. Reescriba la calculadora postfija del capítulo 4 usando **scanf** y/o **sscanf** para hacer la entrada y la conversión. □

7.5 Acceso a archivos

Hasta ahora todos los ejemplos han leído de la entrada estándar y escrito en la salida estándar, las cuales se definen automáticamente para los programas por el sistema operativo local.

El siguiente paso es escribir un programa que dé acceso a un archivo que *no* esté ya conectado al programa. Un programa que ilustra la necesidad de tales operaciones es cat, el cual concatena en la salida estándar un conjunto de archivos nombrados. cat se emplea para escribir archivos en la pantalla, y como un

colector de entradas de propósito general para programas que no disponen de la capacidad de tener acceso a los archivos por nombre. Por ejemplo, la orden

```
cat x.c y.c
```

imprime el contenido de los archivos **x.c** y **y.c** (y nada más) en la salida estándar.

La pregunta es cómo hacer que los archivos nombrados sean leídos —esto es, cómo conectar las proposiciones que leen los datos, con los nombres externos que un usuario tiene en mente.

Las reglas son simples. Antes de que pueda ser leído o escrito, un archivo tiene que ser *abierto* por la función de biblioteca **fopen**, la cual toma un nombre externo como **x.c** o **y.c**, hace algunos arreglos y negociaciones con el sistema operativo (cuyos detalles no deben importarnos), y regresa un apuntador que será usado en posteriores lecturas o escrituras del archivo.

Este apuntador, llamado *apuntador de archivo*, apunta a una estructura que contiene información acerca del archivo, tal como la ubicación de un buffer, la posición de carácter actual en el buffer, si el archivo está siendo leído o escrito y si han ocurrido errores o fin de archivo. Los usuarios no necesitan saber los detalles, debido a que las definiciones obtenidas de <stdio.h> incluyen una declaración de estructura llamada **FILE**. La única declaración necesaria para un apuntador de archivo se ejemplifica por

```
FILE *fp;
FILE *fopen(char *nombre, char *modo);
```

Esto dice que fp es un apuntador a un **FILE**, y **fopen** regresa un apuntador a FILE. Nótese que **FILE** es un nombre de tipo, como int, no un rótulo de estructura; está definido con un **typedef**. (Los detalles de cómo realizar fopen en el sistema UNIX se explican en la sección 8.5.)

La llamada a **fopen** en un programa es

```
fp = fopen(nombre, modo);
```

El primer argumento de **fopen** es una cadena de caracteres que contiene el nombre del archivo. El segundo argumento es el *modo*, también una cadena de caracteres, que indica cómo se intenta emplear el archivo. Los modos disponibles incluyen lectura ("r"), escritura ("w"), y añadido ("a"). Algunos sistemas distinguen entre archivos de texto y binarios; para los últimos, debe escribirse una "b" luego de la cadena de modo.

Si un archivo que no existe se abre para escribir o añadir, se crea, si es posible. Abrir un archivo existente para escribir provoca que los contenidos anteriores sean desechados, mientras que abrirlo para añadir los preserva. Es un error tratar de leer un archivo que no existe, y también pueden haber otras causas de error, como tratar de leer un archivo cuando no se tiene permiso. Si existe cualquier error, fopen regresa NULL. (El error puede ser identificado en forma más precisa; véase la discusión de funciones para manipulación de errores al final de la sección 1 en el apéndice B.)

Lo siguiente que se requiere es una forma de leer o escribir el archivo una vez que está abierto. Existen varias posibilidades, de las cuales **getc** y **putc** son las más simples. **getc** regresa el siguiente carácter de un archivo; necesita el apuntador del archivo para decirle cuál es.

```
int getc(FILE *fp)
```

getc regresa el siguiente carácter del flujo al que se refiere fp; regresa EOF si ocurre algún error.

putc es una función salida:

```
int putc(int c, FILE *fp)
```

putc escribe el carácter c en el archivo fp y regresa el carácter escrito, o EOF si ocurre un error. Tal como getchar y putchar, getc y putc pueden ser macros en lugar de funciones.

Cuando se arranca un programa en C, el medio ambiente del sistema operativo es responsable de abrir tres archivos y proporcionar apuntadores de archivo para ellos. Estos archivos son la entrada estándar, la salida estándar y el error estándar; los apuntadores de archivo correspondientes se llaman **stdin**, **stdout** y **stderr**, y están declarados en <stdio.h>. Normalmente stdin se conecta al teclado y stdout y stderr se conectan a la pantalla, pero stdin y stdout pueden ser redirigidos a archivos o a interconexiones (pipes) como se describe en la sección 7.1.

getchar y putchar pueden estar definidos en términos de getc, putc, stdin y stdout, como sigue:

```
#define getchar( )    getc(stdin)
#define putchar(c)    putc((c), stdout)
```

Para entrada o salida de archivos con formato se pueden emplear las funciones fscanf y fprintf. Estas son idénticas a scanf y printf, excepto en que el primer argumento es un apuntador de archivo que especifica el archivo que será leído o escrito; la cadena de formato es el segundo argumento.

```
int fscanf(FILE *fp, char *formato, ...)
int fprintf(FILE *fp, char *formato, ...)
```

Habiendo hecho a un lado estos prerrequisitos, ya estamos ahora en posición de escribir el programa **cat**, que concatena archivos. El diseño se ha encontrado conveniente para muchos programas. Si existen argumentos en la línea de órdenes, se interpretan como nombres de archivos, y se procesan en orden. Si no hay argumentos, se procesa la entrada estándar.

```
#include <stdio.h>
/* cat: concatena archivos, versión 1 */
main(int argc, char *argv[ ])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
```



```
    if (argc == 1)   /* sin args; copia la entrada estándar
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: no se puede abrir %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy:  copia el archivo ifp al archivo ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}
```

Los apuntadores de archivo stdin y stdout son objetos de tipo FILE *. Sin embargo, son constantes, *no* variables, por lo que no es posible asignarles algo.

La función

```
int fclose(FILE *fp)
```

es lo inverso de fopen; interrumpe la conexión que fue establecida por fopen entre el apuntador de archivo y el nombre externo, liberando al apuntador de archivo para otro archivo. Puesto que la mayoría de los sistemas operativos tienen algunas limitantes sobre el número de archivos que un programa puede tener abiertos simultáneamente, es una buena idea liberar los apuntadores de archivo cuando ya no son necesarios, como se hizo en cat. También hay otra razón para usar fclose en un archivo de salida —vacía el buffer en el cual putc está colectando la salida. Cuando un programa termina normalmente, fclose es llamado automáticamente para cada archivo abierto. (Se puede cerrar stdin y stdout si no son necesarios. También pueden ser reasignados por la función de biblioteca freopen.)

7.6 Manejo de errores—stderr y exit

El manejo de los errores en cat no es el ideal. El problema es que si no se puede tener acceso a uno de los archivos por alguna razón, el diagnóstico se imprime

al final de la salida concatenada. Eso podría ser aceptable si la salida va a la pantalla, pero no si va hacia un archivo o hacia otro programa mediante una interconexión.

Para manejar mejor esta situación, se asigna un segundo flujo de salida, llamado **stderr**, a un programa en la misma forma en que **stdin** y **stdout**. La salida escrita hacia **stderr** normalmente aparece en la pantalla, aun si la salida estándar es redirigida.

Corrijamos **cat** para escribir sus mensajes de error en el archivo de error estándar.

```
#include <stdio.h>

/* cat: concatena archivos, versión 2 */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0];   /* nombre del programa para errores */

    if (argc == 1)   /* sin args; copia la entrada estándar */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                fprintf(stderr, "%s: no se puede abrir %s\n",
                    prog, *argv);
                exit(1)
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: error al escribir stdout\n", prog);
        exit(2);
    }
    exit(0);
}
```

El programa señala errores en dos maneras. Primero, la salida de diagnósticos producida por **fprintf va hacia stderr**, de modo que encuentra su camino hacia la pantalla en lugar de desaparecer en una interconexión o dentro de un archivo de salida. Incluimos el nombre del programa, tomándolo de **argv**[0], en el mensaje, para que si este programa se usa con otros, se identifique la fuente del error.

Segundo, el programa utiliza la función de biblioteca estándar **exit**, que termina la ejecución de un programa cuando se le llama. El argumento de **exit** está disponible para cualquier proceso que haya llamado a éste, para que se pueda probar el éxito o fracaso del programa por otro que lo use como subproceso. Convencionalmente, un valor de retorno 0 señala que todo está bien; los valores diferentes de cero generalmente señalan situaciones anormales. **exit** llama a **fclose** por cada archivo de salida abierto, para vaciar cualquier salida generada a través de un buffer.

Dentro de main, **return expr** es equivalente a **exit**(*expr*). **exit** tiene la ventaja de que puede ser llamada desde otras funciones, y que las llamadas a ella se pueden encontrar con un programa de búsqueda de patrones como el del capítulo 5.

La función **ferror** regresa un valor diferente de cero si ocurrió un error en el flujo **fp**.

```
int ferror(FILE *fp)
```

Aunque los errores de salida son raros, si ocurren (por ejemplo, si un disco se llena), por lo que los programas de producción deben revisar también esto.

La función **feof(FILE *)** es análoga a **ferror**; regresa un valor diferente de cero si ha ocurrido un fin de archivo en el archivo especificado.

```
int feof(FILE *fp)
```

En general, no nos hemos preocupado por el estado de la salida de nuestros pequeños programas ilustrativos, pero todo programa serio debe tener cuidado de regresar valores de estado sensatos y útiles.

7.7 Entrada y salida de líneas

La biblioteca estándar proporciona una rutina de entrada **fgets**, es semejante a la función **getline** que hemos empleado en capítulos anteriores:

```
char *fgets(char *línea, int maxlinea, FILE *fp)
```

fgets lee la siguiente línea (incluyendo el carácter nueva línea) del archivo **fp** y la deja en el arreglo de caracteres línea; se leen hasta **maxline**-1 caracteres. La línea resultante se termina con '\0'. Normalmente, **fgets** regresa línea; en caso de fin de archivo o de error, regresa **NULL**. (Nuestra **getline** regresa la longitud de la línea, que es un valor más útil; cero significa fin de archivo.)

Para salida, la función **fputs** escribe una cadena (que no necesita contener una nueva línea) a un archivo:

```
int fputs(char *línea, FILE *fp)
```

Esta función regresa **EOF** si ocurre un error y cero si no ocurre.

Las funciones de biblioteca **gets** y **puts** son semejantes a **fgets** y **fputs**, pero operan sobre **stdin** y **stdout**. De modo desconcertante, **gets** elimina el '\n' terminal y **puts** lo agrega.

Para mostrar que no hay nada especial sobre funciones como **fgets** y **fputs**, aquí están, copiadas de la biblioteca estándar de nuestro sistema:

```
/* fgets:   obtiene hasta n caracteres de iop */
char *fgets(char *s, int n, FILE *iop)
```

```
        register int c;
        register char *cs;

        cs = s;
        while (--n > 0 && (c = getc(iop)) != EOF)
            if ((*cs++ = c) == '\n')
                break;
        *cs = '\0';
        return (c == EOF && cs == s) ? NULL : s;
}

/* fputs:   coloca la cadena s en el archivo iop */
int fputs(char *s, FILE *iop)
{
        int c;

        while (c = *s++)
            putc(c, iop);
        return ferror(iop) ? EOF : 0;
}
```

Por razones que no son obvias, el estándar especifica valores de retorno diferentes para fgets y fputs.

Es fácil realizar nuestro **getline** a partir de fgets:

```
/* getline:   lee una línea, regresa su longitud */
int getline(char *line, int max)
{
        if (fgets(line, max, stdin) == NULL)
            return 0;
        else
            return strlen(line);
}
```

Ejercicio 7-6. Escriba un programa para comparar dos archivos, imprimiendo la primera línea en donde difieran. □

Ejercicio 7-7. Modifique el programa de búsqueda de un patrón del capítulo 5 para que tome su entrada de un conjunto de archivos nombrados o, si no hay archivos nombrados como argumentos, de la entrada estándar. ¿Debe escribirse el nombre del archivo cuando se encuentra una línea que coincide? □

Ejercicio 7-8. Escriba un programa para imprimir un conjunto de archivos, iniciando cada nuevo archivo en una página nueva, con un título y un contador de página por cada archivo. □

7.8 Otras funciones

La biblioteca estándar proporciona una amplia variedad de funciones. Esta sección es una breve sinopsis de las más útiles. En el apéndice B pueden encontrarse más detalles y muchas otras funciones.

7.8.1 Operaciones sobre cadenas

Ya hemos mencionado las funciones sobre cadenas strlen, strcpy, strcat, y strcmp, que se encuentran en <string.h>. En adelante, s y t son de tipo char *, y c y n son ints.

strcat(s,t)	concatena t al final de s
strncat(s,t,n)	concatena n caracteres de t al final de s
strcmp(s,t)	regresa negativo, cero, o positivo para s < t, s == t, o s > t
strncmp(s,t,n)	igual que strcmp pero sólo en los primeros n caracteres
strcpy(s,t)	copia t en s
strncpy(s,t,n)	copia a lo más n caracteres de t a s
strlen(s)	regresa la longitud de s
strchr(s,c)	regresa un apuntador al primer c que esté en s, o NULL si no está presente
strrchr(s,c)	regresa un apuntador al último c que esté en s, o NULL si no está presente

7.8.2 Prueba y conversión de clases de caracteres

Varias funciones de <ctype.h> realizan pruebas y conversiones de caracteres. En lo que se muestra a continuación, c es un int que se puede representar como un **unsigned char** o **EOF**. Las funciones regresan int.

isalpha(c)	diferente de cero si c es albafética, 0 si no lo es
isupper(c)	diferente de cero si c es mayúscula, 0 si no lo es
islower(c)	diferente de cero si c es minúscula, 0 si no lo es
isdigit(c)	diferente de cero si c es un dígito, 0 si no lo es
isalnum(c)	diferente de cero si isalpha(c) o isdigit(c), 0 si no lo es
isspace(c)	diferente de cero si c es un blanco, tabulador, nueva línea, retorno, avance de línea tabulador vertical
toupper(c)	regresa c convertida a mayúscula
tolower(c)	regresa c convertida a minúscula

7.8.3 Ungetc

La biblioteca estándar proporciona una versión más restringida de la función ungetch que escribimos en el capítulo 4; se llama ungetc.

```
int ungetc(int c, FILE *fp)
```

coloca el carácter c de nuevo en el archivo fp y regresa c, o **EOF** en caso de error. Sólo se garantiza poner un carácter de regreso por archivo. Es posible utilizar ungetc con cualquiera de las funciones como **scanf**, getc o getchar.

7.8.4 Ejecución de órdenes

La función **system(char *s)** ejecuta la orden contenida en la cadena de caracteres **s**, y después continúa la ejecución del programa actual. Los contenidos de s dependen fuertemente del sistema operativo local. Como un ejemplo trivial, en sistemas UNIX, la proposición

```
system("date");
```

provoca que se ejecute el programa **date**, el cual imprime la fecha y hora del día en la salida estándar. **system** regresa del comando ejecutado un estado entero dependiente del sistema. En el sistema UNIX, el estado de retorno es el valor regresado por exit.

7.8.5 Administración del almacenamiento

Las funciones malloc y calloc obtienen bloques de memoria dinámicamente.

```
void  malloc(size_t n)
```

regresa un apuntador a n bytes de almacenamiento no inicializado, o **NULL** si la petición no se satisface.

```
void *calloc(size_t n, size_t size)
```

regresa un apuntador a suficiente espacio para almacenar un arreglo de n objetos del tamaño especificado, o **NULL** si la petición no se satisface. El espacio de almacenamiento es inicializado en cero.

El apuntador regresado por **malloc** o **calloc** tiene la alineación apropiada para el objeto en cuestión, pero se le debe hacer una conversión forzada al tipo apropiado, como en

```
int  ip;

ip = (int *) calloc(n, sizeof(int));
```

free(p) libera el espacio apuntado por **p**, donde p se obtuvo originalmente por una llamada a **malloc** o **calloc**. No existen restricciones sobre el orden en el que se libera el espacio, pero es un grave error el liberar algo no obtenido por una llamada a **calloc** o **malloc**.

También es un error usar algo después de haber sido liberado. Un típico pero erróneo fragmento de código es este ciclo que libera elementos de una lista:

```
for (p = head; p != NULL; p = p->next)    /* INCORRECTO */
    free(p);
```


La forma correcta es guardar lo necesario antes de liberar:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

La sección 8.7 muestra la realización de un asignador de almacenamiento como malloc, en el cual los bloques asignados se pueden liberar en cualquier orden.

7.8.6 Funciones matemáticas

Existen más de veinte funciones matemáticas declaradas en <math.h>; aquí están algunas de las empleadas con más frecuencia. Cada una toma uno o dos argumentos **double** y regresa un **double**.

sin(x)	seno de x, x en radianes
cos(x)	coseno de x, x en radianes
atan2(y,x)	arco tangente de y/x, en radianes
exp(x)	función exponencial e^x
log(x)	logaritmo natural (base e) de x ($x>0$)
log10(x)	logaritmo común (base 10) de x ($x>0$)
pow(x,y)	x^y
sqrt(x,)	raíz cuadrada de x ($x \geq 0$)
fabs(x)	valor absoluto de x

7.8.7 Generación de números aleatorios

La función rand( ) calcula una secuencia de enteros pseudoaleatorios en el rango de cero a **RAND_MAX**, que está definido en <stdlib.h>. Una forma de producir números aleatorios de punto flotante mayores o iguales a cero pero menores que uno es

```
#define frand( ) ((double) rand( ) / (RAND_MAX+1))
```

(Si su biblioteca ya proporciona una función para números aleatorios de punto flotante, es probable que tenga mejores propiedades estadísticas que ésta.)

La función **srand(unsigned)** fija la semilla para rand. La implantación portátil de **rand** y de **srand** sugerida por el estándar aparece en la sección 2.7.

Ejercicio 7-9. Se pueden escribir funciones como **isupper** para ahorrar espacio o tiempo. Explore ambas posibilidades. □

CAPITULO 8: La interfaz con el sistema UNIX

El sistema operativo UNIX proporciona sus servicios a través de un conjunto de *llamadas al sistema*, que consisten en funciones que están dentro del sistema operativo y que pueden ser invocadas por programas del usuario. Este capítulo describe cómo emplear algunas de las más importantes llamadas al sistema desde programas en C. Si el lector usa UNIX, esto debe serle directamente útil, debido a que algunas veces es necesario emplear llamadas al sistema para tener máxima eficiencia, o para tener acceso a alguna facilidad que no esté en la biblioteca. Incluso, si se emplea C en un sistema operativo diferente el lector debería ser capaz de adentrarse en la programación estudiando estos ejemplos; aunque los detalles varían, se encontrará un código semejante en cualquier sistema. Puesto que la biblioteca de C ANSI está en muchos casos modelada con base en las facilidades de UNIX, este código puede ayudar también a su entendimiento.

El capítulo está dividido en tres partes fundamentales: entrada/salida, sistema de archivos y asignación de almacenamiento. Las primeras dos partes suponen una modesta familiaridad con las características externas de los sistemas UNIX.

El capítulo 7 tuvo que ver con una interfaz de entrada/salida uniforme entre sistemas operativos. En cualquier sistema las rutinas de la biblioteca estándar se tienen que escribir en términos de las facilidades proporcionadas por el sistema anfitrión. En las secciones de este capítulo describiremos las llamadas al sistema UNIX para entrada y salida, y mostraremos cómo puede escribirse parte de la biblioteca estándar con ellas.

8.1 Descriptores de archivos

En el sistema operativo UNIX, todas las entradas y salidas se realizan por la lectura o escritura de archivos, debido a que los dispositivos periféricos, aun el teclado y la pantalla, son archivos que están en el sistema. Esto significa que una sencilla interfaz homogénea maneja todas las comunicaciones entre un programa y los dispositivos periféricos.

En el caso más general, antes de leer o escribir un archivo, primero se debe informar al sistema acerca de la intención de hacerlo, mediante el proceso llamado

abrir un archivo. Si se va a escribir en un archivo también puede ser necesario crearlo o descartar el contenido previo. El sistema verifica los derechos de hacer tal cosa (¿El archivo existe? ¿tiene permiso de hacer acceso a él?) y, si todo está correcto, regresa al programa un pequeño entero no negativo llamado *descriptor de archivo*. Siempre que se van a efectuar acciones de entrada y salida sobre ese archivo, se usa el descriptor de archivo para identificarlo en lugar del nombre. (Un descriptor de archivo es análogo al apuntador de archivo usado por la biblioteca estándar o al manipulador de archivo de MS-DOS.) Toda la información acerca de un archivo abierto es mantenida por el sistema; el programa del usuario se refiere al archivo sólo por el descriptor.

Puesto que es tan común que la entrada y la salida involucren al teclado y a la pantalla, existen arreglos especiales para hacer esto convenientemente. Cuando el intérprete de comandos (el "shell") ejecuta un programa se abren tres archivos, con descriptores 0, 1, 2, llamados entrada estándar, salida estándar y error estándar. Si un programa lee de 0 y escribe a 1 y a 2, puede hacer entrada y salida sin preocuparse de abrir archivos.

El usuario de un programa puede redirigir la E/S hacia y desde archivos con < y >:

```
prog <archent >archsal
```

En este caso, shell cambia las asignaciones predefinidas para los descriptores 0 y 1 a los archivos nombrados. Normalmente el descriptor de archivo 2 permanece asignado a la pantalla, para que los mensajes de error puedan ir hacia allá. Observaciones semejantes se aplican para la entrada y salida asociada con una interconexión. En todos los casos, la asignación de archivos la cambia shell, no el programa. El programa no sabe de dónde proviene su entrada ni hacia dónde va su salida, mientras use al archivo 0 para entrada y 1 y 2 para salida.

8.2 E/S de bajo nivel—read y write

La entrada y salida usa las llamadas al sistema **read** y **write**, a las que se tiene acceso desde programas escritos en C a través de dos funciones llamadas **read** y **write**. Para ambas, el primer argumento es un descriptor de archivo. El segundo argumento es un arreglo de caracteres perteneciente al programa hacia o de donde los datos van a ir o venir. El tercer argumento es el número de bytes que serán transferidos.

```
int n_leídos = read(int fd, char *buf, int n);
int n_escritos = write(int fd, char *buf, int n);
```

Cada llamada regresa una cuenta del número de bytes transferidos. En la lectura, el número de bytes regresados puede ser menor que el número solicitado. Un valor de regreso de cero bytes implica fin de archivo y −1 indica un error de algún

tipo. Para escritura, el valor de retorno es el número de bytes escritos; si éste no es igual al número solicitado, ha ocurrido un error.

En una llamada pueden leerse cualquier número de bytes. Los valores más comunes son 1, que significa un carácter a la vez (sin buffer), y un número como 1024 o 4098, que corresponde al tamaño de un bloque físico de un dispositivo periférico. Los valores mayores serán más eficientes debido a que serán realizadas menos llamadas al sistema.

Para juntar estos temas, podemos escribir un sencillo programa que copie su entrada a su salida, el equivalente del programa copiador de archivos escrito para el capítulo 1. Este programa copiará cualquier cosa a cualquier cosa, ya que la entrada y la salida pueden ser redirigidas hacia cualquier archivo o dispositivo.

```
#include "syscalls.h"

main()   /* copia la entrada a la salida */
{
     char buf(BUFSIZ);
     int n;

     while ((n = read(0, buf, BUFSIZ)) > 0)
          write(1, buf, n);
     return 0;
}
```

Hemos reunido prototipos de funciones para las llamadas al sistema en un archivo llamado **syscalls.h**, de modo que podamos incluirlo en los programas de este capítulo. Sin embargo, este nombre no es estándar.

El parámetro **BUFSIZ** también está definido dentro de **syscalls.h**; su valor es un tamaño adecuado para el sistema local. Si el tamaño del archivo no es un múltiplo de **BUFSIZ**, algún **read** regresará un número menor de bytes a ser escritos por **write**; la siguiente llamada a **read** después de eso regresará cero.

Es instructivo ver cómo se pueden usar **read** y **write** para construir rutinas de alto nivel como **getchar**, **putchar**, etc. Por ejemplo, aquí está una versión de **getchar** que realiza entrada sin buffer, leyendo de la entrada estándar un carácter a la vez.

```
#include "syscalls.h"

/* getchar: entrada de un carácter simple sin buffer */
int getchar(void)
{
     char c;

     return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
}
```

c debe ser un **char**, a que **read** necesita un apuntador a carácter. Forzar **c** a ser **unsigned char** en la proposición de regreso elimina cualquier problema de extensión de signo.

La segunda versión de **getchar** hace la entrada en grandes fragmentos y saca los caracteres uno a la vez.

```
#include "syscalls.h"
/* getchar:    versión con buffer simple */
int getchar(void)
{
     static char buf[BUFSIZ];
     static char *bufp = buf;
     static int n = 0;

     if (n == 0) {   /* el buffer está vacío */
          n = read(0, buf, sizeof buf);
          bufp = buf;
     }
     return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Si esta versión de **getchar** fuese a ser compilada con <stdio.h> incluida, sería necesario eliminar la definición del nombre **getchar** con **#undef** en caso de que esté implantada como una macro.

8.3 Open, creat, close, unlink

Además de la entrada, la salida y el error estándar, se pueden abrir explícitamente archivos para leerlos o escribirlos. Existen dos llamadas al sistema para esto, **open** y **creat**.§

open es como el **fopen** expuesto en el capítulo 7, excepto que en lugar de regresar un apuntador de archivo, regresa un descriptor de archivo, que es tan sólo un **int**. **open** regresa —1 si ocurre algún error.

```
#include <fcntl.h>

int fd;
int open(char *nombre, int flags, int perms);

fd = open(nombre, flags, perms);
```

Como con **fopen**, el argumento **nombre** es una cadena de caracteres que contiene el nombre del archivo. El segundo argumento, **flags**, es un **int** que especifica cómo será abierto el archivo; los principales valores son

§ A pesar de que en inglés la palabra correcta es "**create**", el nombre de la función es sólo "**creat**". (N. de T.)

O_RDONLY	abrir sólo para lectura
O_WRONLY	abrir sólo para escritura
O_RDWR	abrir para lectura y escritura

Estas constantes están definidas en <fcntl.h> en sistemas UNIX System V, y en <sys/file.h> en versiones Berkeley (BSD).

Para abrir un archivo ya existente para lectura,

```
fd = open(nombre, O_RDONLY, 0);
```

El argumento **perms** es siempre cero para los usos de **open** que discutiremos.

Es un error tratar de abrir un archivo que no existe. Para crear nuevos archivos o reescribir anteriores, se proporciona la llamada al sistema **creat**.

```
int creat(char *nombre, int perms);

fd = creat(nombre, perms);
```

regresa un descriptor de archivo si fue capaz de crear el archivo, y −1 si no lo fue. Si el archivo ya existe, **creat** lo truncará a longitud cero y por tanto descartará su contenido previo; no es un error crear creat un archivo que ya existe.

Si el archivo no existe, **creat** lo crea con los permisos especificados por el argumento **perms**. En el sistema de archivos de UNIX hay nueve bits para información de permisos asociados con un archivo, que controlan el acceso a la lectura, escritura y ejecución para el propietario del archivo, para el grupo del propietario y para todos los demás. Así, un número octal de tres dígitos es conveniente para especificar los permisos. Por ejemplo, 0755 especifica permisos para leer, escribir y ejecutar para el propietario, y leer y ejecutar para el grupo y para cualquier otro.

Para ilustrarlo, aquí está una versión simplificada del programa **cp** de UNIX, que copia un archivo a otro. Nuestra versión copia sólo un archivo, no permite que el segundo argumento sea un directorio e inventa los permisos en lugar de copiarlos.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666   /* lectura y escritura para propietario, grupo y otros */

void error(char *, ...);

/* cp:   copia f1 a f2 */
main(int argc, char *argv[])
{
     int f1, f2, n;
     char buf[BUFSIZ];

     if (argc != 3)
          error ("Uso: cp de hacia");
```

```
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: no se puede abrir %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: no se puede crear %s, modo %03o",
            argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error ("cp: error de escritura en el archivo %s", argv[2]);
    return 0;
}
```

Este programa crea el archivo de salida con permisos fijos 0666. Con la llamada al sistema stat, descrita en la sección 8.6, podemos determinar el modo de un archivo existente y así dar el mismo modo a la copia.

Nótese que la función error es invocada con una lista variable de argumentos muy semejante a la de printf. La realización de error ilustra cómo utilizar otros miembros de la familia printf. La función de biblioteca estándar vprintf es como printf, excepto que la lista variable de argumentos es reemplazada por un solo argumento que ha sido inicializado llamando a la macro va_start. En forma semejante, vfprintf y vsprintf coinciden con fprintf y sprintf.

```
#include <stdio.h>
#include <stdarg.h>

/* error:  imprime un mensaje de error y muere */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

Existe un límite (regularmente 20) en el número de archivos que un programa puede tener abiertos simultáneamente. De acuerdo con esto, un programa que intente procesar muchos archivos debe ser preparado para reutilizar descriptores de archivo. La función close(int fd) suspende la conexión entre un descriptor de archivo y un archivo abierto, y libera al descriptor de archivo para ser utilizado con algún otro archivo; corresponde a fclose de la biblioteca estándar excepto en que no existe un buffer que vaciar. La terminación de un programa vía exit o return desde el programa principal cierra todos los archivos abiertos.

La función unlink(char *nombre) remueve el archivo nombre del sistema de archivos. Corresponde a la función de la biblioteca estándar **remove**.

Ejercicio 8-1. Reescriba el programa cat del capítulo 7 usando read, write, open y close, en lugar de sus equivalentes de la biblioteca estándar. Haga experimentos para determinar la velocidad relativa de las dos versiones. □

8.4 Acceso aleatorio—lseek

La entrada y la salida son normalmente secuenciales: cada read o write ocurre en una posición del archivo justo después de la anterior. Sin embargo, cuando es necesario, un archivo se puede leer o escribir en cualquier orden arbitrario. La llamada al sistema lseek proporciona una forma de moverse en un archivo sin leer o escribir ningún dato:

```
long lseek(int fd, long offset, int origen);
```

fija en offset la posición actual en el archivo cuyo descriptor es fd, que se toma relativo a la localización especificada por origen. Una lectura o escritura posterior principiará en esa posición. origen puede ser 0, 1 o 2 para especificar que el desplazamiento offset será medido desde el principio, desde la posición actual, o desde el fin del archivo, respectivamente. Por ejemplo, para agregar a un archivo (la redirección >> en el shell de UNIX, o "a" de fopen), hay que ir al final antes de escribir:

```
lseek(fd, 0L, 2);
```

Para regresar al principio ("rebobinar"),

```
lseek(fd, 0L, 0);
```

Nótese el argumento 0L; también podría ser escrito como (long) 0 o sólo como 0 si lseek está declarado adecuadamente.

Con lseek, es posible tratar a los archivos más o menos como arreglos extensos, al precio de un acceso más lento. Por ejemplo, la siguiente función lee cualquier número de bytes en cualquier lugar arbitrario de un archivo. Regresa el número leído, o -1 en caso de error.

```
#include "syscalls.h"

/* get:  lee n bytes de la posición pos */
int get(int fd, long pos, char *buf, int n)
{
    if (lseef(fd, pos, 0) >= 0)  /* se sitúa en pos */
        return read(fd, buf, n);
```

```
        else
            return -1;
}
```

El valor de regreso de **lseek** es un long que da la nueva posición en el archivo, o −1 si ocurre un error. La función de biblioteca estándar **fseek** es semejante a lseek, excepto en que el primer argumento es un FILE * y el valor de regreso es diferente de cero si ocurrió un error.

8.5 Ejemplo—una realización de fopen y getc

Ilustremos ahora cómo algunas de estas piezas quedan juntas, mostrando una realización de las rutinas **fopen** y **getc** de la biblioteca estándar.

Recuérdese que los archivos en la biblioteca estándar son descritos por apuntadores de archivos en vez de con descriptores de archivo. Un apuntador de archivo es un apuntador a una estructura que contiene información varia acerca del archivo: un apuntador a un buffer, para que el archivo pueda ser leído en grandes fragmentos; una cuenta del número de caracteres que quedan en el buffer; un apuntador a la posición del siguiente carácter en el buffer; el descriptor de archivo, y banderas que describen el modo de lectura/escritura, estado de error, etcétera.

La estructura de datos que describe un archivo está contenida en **<stdio.h>**, que se debe incluir (con **#include**) en cualquier archivo fuente que utilice rutinas de la biblioteca de entrada/salida estándar. También está incluido en las funciones de la biblioteca. En el siguiente fragmento de un **<stdio.h>** típico, los nombres que se intenta emplear sólo en las funciones de la biblioteca estándar principian con un subguión, por lo que son menos susceptibles de tener conflicto con nombres en los programas del usuario. Esta convención la emplean todas las rutinas de la biblioteca estándar.

```
#define NULL        0
#define EOF         (-1)
#define BUFSIZ      1024
#define OPEN_MAX    20 /* máximo número de archivos abiertos a la vez */

typedef struct _iobuf {
    int cnt;            /* caracteres que quedan */
    char *ptr;          /* posición del siguiente carácter */
    char *base;         /* localización del buffer */
    int flag;           /* modo de acceso al archivo */
    int fd;             /* descriptor de archivo */
} FILE;
extern FILE _iob[OPEN_MAX];
```



```
#define stdin       (&_iob[0])
#define stdout      (&_iob[1])
#define stderr      (&_iob[2])

enum _flags {
    _READ   = 01,   /* archivo abierto para lectura */
    _WRITE  = 02,   /* archivo abierto para escritura */
    _UNBUF  = 04,   /* archivo sin buffer */
    _EOF    = 010,  /* ocurrió fin de archivo (EOF) en este archivo */
    _ERR    = 020   /* ocurrió un error en este archivo */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)     (((p)->flag & _EOF) != 0)
#define ferror(p)   (((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->fd)

#define getc(p)     (--(p)->cnt >= 0 \
                ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p)   (--(p)->cnt >= 0 \
                ? (p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()   getc(stdin)
#define putchar(x)  putc((x), stdout)
```

La macro **getc** normalmente decrementa la cuenta, avanza el apuntador y regresa el carácter. (Recuerde que un **#define** largo se continúa con una diagonal invertida.) Si la cuenta se hace negativa, sin embargo, **getc** llama a la función **_fillbuf** para llevar el buffer, reinicializa el contenido de la estructura, y regresa un carácter. Los caracteres son devueltos **unsigned**, lo que asegura que todos los caracteres serán positivos.

Aunque no discutiremos ningún detalle, hemos incluido la definición de putc para mostrar que opera en forma muy semejante a **getc**, llamando a una función _flushbuf cuando su buffer está lleno. También hemos incluido macros para tener acceso al estado de error, fin de archivo, y al descriptor del mismo.

Ahora puede escribirse la función **fopen**. La mayor parte de **fopen** tiene que ver con tener el archivo abierto y colocado en el lugar correcto, y con fijar los bits de la bandera **flag** para indicar el estado apropiado. **fopen** no asigna ningún espacio para el buffer; esto es realizado por **_fillbuf** cuando el archivo se lee por primera vez.

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666  /* lectura y escritura para propietario, grupo, otros */
```

```
/* fopen: abre un archivo, regresa un apuntador de archivo */
FILE *fopen(char *name, char *mode)
{
    int fd;
    FILE *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break;          /* se encontró una entrada libre */
    if (fp >= _iob + OPEN_MAX)      /* no hay entradas libres */
        return NULL;

    if (*mode == 'w')
        fd = creat(name, PERMS);
    else if (*mode == 'a') {
        if ((fd = open(name, O_WRONLY, 0)) == -1)
            fd = creat(name, PERMS);
        lseek(fd, 0L, 2);
    } else
        fd = open(name, O_RDONLY, 0);
    if (fd == -1)           /* no hubo acceso al nombre */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mode == 'r') ? _READ : _WRITE;
    return fp;
}
```

Esta versión de **fopen** no maneja todas las posibilidades de modos de acceso del estándar, aunque el agregarlas no se llevaría mucho código. En particular, nuestra **fopen** no reconoce la "b" que indica acceso binario, ya que eso no tiene significado en sistemas UNIX, ni el "+" que permite tanto lectura como escritura.

La primera llamada a **getc** para un archivo en particular encuentra una cuenta de cero, lo que obliga a una llamada a **_fillbuf**. Si **_fillbuf** encuentra que el archivo no está abierto para lectura, regresa **EOF** de inmediato. De otra forma, trata de asignar un buffer (si la lectura será con buffer).

Una vez que el buffer ha sido establecido, **_fillbuff** llama a read para llenarlo, fija la cuenta y los apuntadores, y regresa el carácter del principio del buffer. Las posteriores llamadas a **_fillbuf** encontrarán un buffer asignado.


```
#include "syscalls.h"

/* _fillbuf:   asigna y llena un buffer de entrada */
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag&(_READ|_EOF|_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL)     /* sin buffer aún */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF;       /* no puede obtener un buffer */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}
```

El único cabo suelto es cómo arrancar todo. El arreglo **_iob** debe ser definido e inicializado para **stdin**, **stdout** y **stderr**:

```
FILE _iob[OPEN_MAX] = {     /* stdin, stdout, stderr: */
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 }
};
```

La inicialización de la parte **flag** de la estructura muestra que **stdin** será leído, **stdout** será escrito, y **stderr** será escrito sin buffer.

Ejercicio 8-2. Reescriba **fopen** y **_fillbut** con campos en vez de operaciones explícitas de bits. Compare el tamaño del código y la velocidad de ejecución. □

Ejercicio 8-3. Diseñe y escriba **_flushbuf**, **fflush**, y **fclose**. □

Ejercicio 8-4. La función de biblioteca estándar

```
int fseek(FILE *fp, long offset, int origen)
```

es idéntica a **lseek** excepto que **fp** es un apuntador de archivo en vez de un descriptor de archivo, y el valor regresado es un estado **int**, no una posición. Escriba **fseek**. Asegúrese de que su **fseek** se coordina apropiadamente con el manejo de buffers realizado por las otras funciones de la biblioteca. □

8.6 Ejemplo—listado de directorios

Algunas veces se requiere una forma diferente de interacción con el sistema de archivos, para determinar información *acerca* de un archivo, no lo que contiene. Un programa que lista un directorio tal como la orden **ls** de UNIX es un ejemplo —imprime los nombres de los archivos que están en el directorio, y, en forma optativa, más información, tal como tamaños, permisos y esas cosas. La orden dir de MS-DOS es análoga.

Como un directorio de UNIX es simplemente un archivo, **ls** sólo necesita leerlo para obtener los nombres de archivos. Pero es necesario utilizar una llamada al sistema para tener acceso a la otra información acerca del archivo, tal como su tamaño. En otros sistemas puede ser necesaria una llamada al sistema incluso para los nombres de los archivos; éste es el caso de MS-DOS, por ejemplo. Lo que nosotros queremos es proporcionar acceso a la información en una forma relativamente independiente del sistema, a pesar incluso de que la realización pueda ser altamente dependiente del sistema.

Ilustraremos algo de esto escribiendo un programa llamado **fsize**. **fsize** es una forma especial de **ls** que imprime los tamaños de todos los archivos nombrados en su lista de argumentos. Si uno de los archivos es un directorio **fsize** se aplica en forma recursiva para ese directorio. Si no hay ningún argumento, procesa el directorio actual.

Principiemos con una breve revisión de la estructura del sistema de archivos de UNIX. Un *directorio* es un archivo que contiene una lista de nombres de archivo y algunas indicaciones de dónde se localizan. La "localización" es un índice en otra tabla llamada la "lista de nodos-i". El *nodo-i* para un archivo es donde se mantiene toda la información acerca de un archivo, excepto su nombre. Una entrada en el directorio consiste generalmente en sólo dos ítems, el nombre del archivo y el número de nodo-i.

Desafortunadamente, el formato y el contenido preciso de un directorio no es el mismo en todas las versiones del sistema. De modo que dividiremos la tarea en dos partes para tratar de aislar las partes no transportables. El nivel más externo define una estructura llamada **Dirent** y tres rutinas, **opendir**, **readdir**, y **closedir** para proporcionar acceso independiente del sistema al nombre y número de nodo-i en una entrada del directorio. Escribiremos **fsize** con esta interfaz. Después mostraremos cómo hacer esto en sistemas que usan la misma estructura de directorios que UNIX Versión 7, y System V; las variantes son dejadas como ejercicios.

La estructura **Dirent** contiene el número de nodo-i y el nombre. La longitud máxima de un componente del nombre de archivo es **NAME_MAX**, que es un valor

dependiente del sistema. **opendir** regresa un apuntador a una estructura llamada **DIR**, análoga a **FILE**, que es empleada por **readdir** y **closedir**. La información es recolectada en un archivo llamado **dirent.h**.

```
#define NAME_MAX   14  /* componente de nombre de archivo más gran-
                          de; */
                       /* dependiente del sistema */

typedef struct {    /* entrada de directorio transportable: */
    long ino;                 /* número de nodo-i */
    char name[NAME_MAX+1];    /* nombre + terminador '\0' */
} Dirent;

typedef struct {       /* DIR mínima: sin buffer, etc. */
    int fd;              /* descriptor de archivo para el directorio */
    Dirent d;            /* la entrada del directorio */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

La llamada al sistema **stat** toma un nombre de archivo y regresa toda la información que está en el nodo-i para ese archivo, o −1 si existe un error. Esto es,

```
char *nombre;
struct stat stbuf;
int stat(char *, struct stat *);

stat(nombre, &strbuf);
```

llena la estructura **stbuf** con la información del nodo-i para el nombre de archivo. La estructura que describe el valor regresado por **stat** está en <sys/stat.h>, y típicamente se ve así:

```
struct stat    /* información de nodo-i regresada por stat */
{
    dev_t    st_dev;     /* dispositivo de nodo -i */
    ino_t    st_ino;     /* número de nodo-i */
    short    st_mode;    /* bits de modo */
    short    st_nlink;   /* número de ligas al archivo */
    short    st_uid;     /* id. de usuario del propietario */
    short    st_gid;     /* id. de grupo del propietario */
    dev_t    st_rdev;    /* para archivos especiales */
    off_t    st_size;    /* tamaño del archivo en caracteres */
    time_t   st_atime;   /* hora del último acceso */
    time_t   st_mtime;   /* hora de la última modificación */
    time_t   st_ctime;   /* hora de creación original */
};
```

La mayoría de estos valores son explicados por los campos de comentario. Los tipos como dev_t y ino_t están definidos en <sys/types.h>, que también debe ser incluido.

La entrada st_mode contiene un conjunto de banderas que describen el archivo. La definición de banderas está también incluida en <sys/stat.h>; sólo requerimos de la parte que tiene que ver con el tipo de archivo

```
#define   S_IFMT 0160000    /* tipo de archivo */
#define   S_IFDIR 0040000   /* directorio */
#define   S_IFCHR 0020000   /* especial de caracteres */
#define   S_IFBLK 0060000   /* especial de bloques */
#define   S_IFREG 0100000   /* regular */
   /* ... */
```

Ahora estamos listos para escribir el programa fsize. Si el modo obtenido de stat indica que un archivo no es un directorio, entonces el tamaño está a la mano y puede ser impreso directamente. Si el archivo es un directorio, sin embargo, entonces tenemos que procesar ese directorio un archivo a la vez; puede a su vez contener subdirectorios, de modo que el proceso es recursivo.

La rutina principal trata con los argumentos de la línea de órdenes; pasa cada argumento a la función fsize.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>         /* banderas para lectura y escritura */
#include <sys/types.h>     /* typedefs */
#include <sys/stat.h>      /* estructura regresada por stat */
#include "dirent.h"

void fsize(char *);

/* imprime tamaños de archivos */
main(int argc, char **argv)
{
    if (argc == 1)         /* default: directorio actual */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

La función fsize imprime el tamaño del archivo. Sin embargo, si el archivo es un directorio, fsize llama primero a dirwalk para manejar todos los archivos en él. Note como se usan los nombres de las banderas S_IFMT y S_IFDIR de <sys/stat.h> para decidir si el archivo es un directorio. El uso de los paréntesis importa, debido a que la precedencia de & es inferior que la de ==.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn) (char *));
```



```
/* fsize:   imprime el tamaño del archivo "name" */
void fsize(char *name)
{
    struct stat atbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: no se tiene acceso a %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

La función dirwalk es una rutina de propósito general que aplica una función a cada archivo que está dentro de un directorio. Abre el directorio, itera con todos los archivos que hay en él, llamando en cada uno a la función; después cierra el directorio y regresa. Puesto que fsize llama a dirwalk en cada directorio, las dos funciones se llaman recursivamente una a la otra.

```
#define MAX_PATH 1024

/* dirwalk:   aplica fcn a todos los archivos de dir */
void dirwalk(char *dir, void (*fcn) (char *))
{
    char name[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: no se puede abrir %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->name, ".") == 0
        || strcmp(dp->name, ".") == 0)
            continue;    /* se ignora a si mismo y a su padre */
        if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
            fprintf(stderr, "dirwalk:nombre %s/%s demasiado largo\n",
                dir, dp->name);
        else {
            sprintf(name, "%s/%s", dir, dp->name);
            (*fcn) (name);
        }
    }
    closedir(dfd);
}
```

Cada llamada a **readdir** regresa un apuntador a información para el siguiente archivo, o NULL cuando ya no quedan archivos. Cada directorio siempre contiene entradas para sí mismo, llamada ".", y para su padre ".."; deben ser ignoradas, o el programa iterará por siempre.

En este nivel, el código es independiente de cómo está el formato de los directorios. El siguiente paso es presentar versiones mínimas de **opendir, readdir,** y **closedir** para un sistema específico. Las siguientes rutinas son para sistemas UNIX Versión 7 y System V; utilizan la información que está en el *header* **<sys/dir.h>**, que aparece así:

```
#ifndef DIRSIZ
#define DIRSIZ    14
#endif
struct direct   /* entrada del directorio  /
{
    ino_t d_ino;                /* número de nodo-i */
    char d_name[DIRSIZ];   /* los nombres largos no tienen '\0' */
};
```

Algunas versiones del sistema permiten nombres mucho más largos y tienen una estructura de directorios más complicada.

El tipo **ino_t** es un **typedef** que describe al índice a la lista de nodos-i. En el sistema que usamos regularmente es un **unsigned short**, pero ésta no es la clase de información para incluir en un programa; puede ser distinta en un sistema diferente, de modo que **typedef** es mejor. Un juego completo de tipos "del sistema" se encuentra en **<sys/types.h>**.

opendir abre el directorio, verifica que el archivo sea un directorio (esta vez por medio de la llamada al sistema **fstat**, que es como **stat** excepto en que se aplica a un descriptor de archivo), asigna una estructura de directorio, y graba la información.

```
int fstat(int fd, struct stat *);

/* opendir:  abre un directorio para llamadas de readdir */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
     || fstat(fd, &stbuf) == -1
     || (stbuf.st_mode & S_IFMT) != S_IFDIR
     || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
          return NULL;
```



```
    dp->fd = fd;
    return dp;
}
```

closedir cierra el archivo del directorio y libera el espacio:

```
/* closedir:  cierra un directorio abierto por opendir */
void closedir(DIR *dp)
{
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Finalmente, **readdir** usa a **read** para leer cada entrada del directorio. Si una entrada del directorio no está actualmente en uso (debido a que ha sido removido un archivo), el número de nodo-i es cero, y esta posición se salta. De otra forma, el número de nodo-i y el nombre son colocados en una estructura estática y se regresa al usuario un apuntador a ella. Cada llamada sobreescribe la información de la anterior.

```
#include <sys/dir.h>      /* estructura local de directorio */

/ readdir:     lee en secuencia las entradas de un directorio */
Dirent  readdir(DIR *dp)
{
    struct direct dirbuf; /  estructura local de directorio */
    static Dirent d;       /  regreso: estructura transportable */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
                == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0)    /* entrada que no está en uso */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0'; /* asegura la terminación */
        return &d;
    }
    return NULL;
}
```

Aunque el programa **fsize** es bastante especializado, ilustra un par de ideas importantes. Primera, muchos programas no son "programas del sistema"; simplemente usan información que es mantenida por el sistema operativo. Para tales programas es crucial que la representación de la información aparezca sólo en *headers* estándar, y que los programas incluyan esos archivos en vez de tener

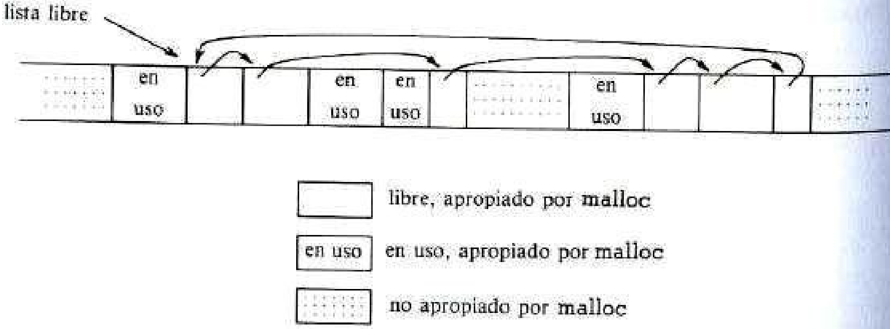
las declaraciones en ellos mismos. La segunda observación es que con cuidado es posible crear una interfaz hacia objetos dependientes del sistema que a su vez sea relativamente independiente del mismo. Las funciones de la biblioteca estándar son buenos ejemplos.

Ejercicio 8-5. Modifique el programa **fsize** para que imprima el resto de la información contenida en la entrada del nodo-i. □

8.7 Ejemplo—asignador de memoria

En el capítulo 5 presentamos un asignador de memoria muy limitado que funcionaba en modo de pila. La versión que escribiremos ahora no tiene restricciones. Las llamadas a **malloc** y **free** pueden ocurrir en cualquier orden; **malloc** llama al sistema operativo para obtener más memoria cuando es necesaria. Estas rutinas ilustran algunas de las consideraciones implicadas en la creación de código dependiente de máquina en una forma relativamente independiente, y también muestran una aplicación de estructura, uniones y **typedef** a la vida real.

En vez de asignar un arreglo precompilado de tamaño fijo, **malloc** solicitará espacio al sistema operativo cuando sea necesario. Dado que otras actividades en el programa también pueden requerir espacio sin llamar a este asignador, el espacio que **malloc** maneja puede no ser contiguo. Así, el espacio libre de almacenamiento es mantenido como una lista de bloques libres. Cada bloque contiene un tamaño, un apuntador al siguiente bloque, y el espacio en sí. Los bloques son mantenidos en orden ascendente de dirección de almacenamiento, y el último bloque (dirección más alta) apunta al primero.



Cuando se hace una solicitud, se rastrea la lista libre hasta que se encuentra un bloque suficientemente grande. Este algoritmo es llamado "de primer ajuste" (*first-fit*), en contraste con (*best fit*), que busca el bloque más pequeño que satisfará la solicitud. Si el bloque es exactamente del tamaño requerido, se desliga de la lista y se entrega al usuario. Si el bloque es demasiado grande se divide, y la cantidad apropiada es entregada al usuario mientras que el resto permanece en la lista libre. Si no se encuentra un bloque suficientemente grande, algún otro trozo grande se obtiene del sistema operativo y se liga a la lista libre.

La liberación también provoca una búsqueda en la lista libre, para encontrar el lugar apropiado para insertar el bloque que está siendo liberado. Si el bloque que está siendo liberado es adyacente a un bloque libre en cualquiera de sus lados, se une con él en un bloque único más grande, por lo que el almacenamiento no se fragmenta demasiado. Determinar la adyacencia es fácil puesto que la lista libre es mantenida en orden ascendente de direcciones.

Un problema, al que aludimos en el capítulo 5, es asegurar que el almacenamiento regresado por **malloc** esté alineado apropiadamente para los objetos que se almacenarán en él. Aunque las máquinas varían, para cada una existe un tipo que es el más restrictivo: si el tipo más restrictivo puede ser almacenado en una dirección particular, todos los otros tipos también lo serán. En algunas máquinas, el tipo más restrictivo es un **double**; en otras, basta **int** o **long**.

Un bloque libre contiene un apuntador al siguiente bloque de la cadena, un registro del tamaño del bloque, y luego el espacio disponible en sí; la información de control que está al inicio es llamada el encabezador. Para simplificar la alineación, todos los bloques son múltiplos del tamaño del encabezador, y éste se alinea apropiadamente. Esto se logra mediante una unión que contiene la estructura deseada del encabezador y una ocurrencia del tipo de alineación más restrictivo, al que arbitrariamente hemos hecho **long**:

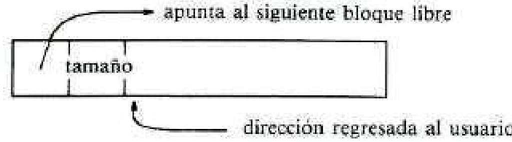
```
typedef long Align;   /* para alineamiento al límite mayor */

union header {          /* encabezador del bloque */
    struct {
        union header *ptr; /* siguiente bloque si está en la lista libre */
        unsigned size;     /* tamaño de este bloque */
    } s;
    Align x;            /* obliga a la alineación de bloques */
};

typedef union header Header;
```

El campo Align nunca es utilizado; sólo hace que cada encabezador esté alineado al límite del peor caso.

En **malloc**, el tamaño requerido en caracteres es redondeado al número apropiado de unidades de tamaño del encabezador; el bloque que será asignado contiene una unidad más, para el encabezador en sí, y éste es el valor grabado en el campo **size**. El apuntador es regresado por **malloc** apunta al espacio libre, no encabezador. El usuario puede hacer cualquier cosa con el espacio requerido, pero si algo se escribe fuera del espacio asignado, la lista se puede desorganizar.



Un bloque regresado por **malloc**

El campo size es necesario debido a que los bloques controlados por **malloc** no requieren ser contiguos —no es posible calcular tamaños mediante aritmética de apuntadores.

La variable base se usa para comenzar. Si **freep** es NULL, como lo es en la primer llamada de **malloc**, entonces se crea una lista libre degenerada que contiene un bloque de tamaño cero y apunta a sí misma. En cualquier caso, luego se busca en la lista libre. La búsqueda de un bloque libre de tamaño adecuado principia en el punto **(freep)** donde se encontró el último bloque; esta estrategia ayuda a mantener la lista homogénea. Si se encuentra un bloque demasiado grande, al usuario se le regresa la parte final; en esta forma el encabezador del original sólo necesita tener ajustado su tamaño. En todos los casos, el apuntador regresado al usuario apunta al espacio libre dentro del bloque, que principia una unidad más allá del encabezador.

```
static Header base;          /* lista vacía para iniciar */
static Header *freep = NULL;     /* inicio de una lista libre */

/* malloc: asignador de almacenamiento de propósito general */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) { /* no hay lista libre aún */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {      /* suficientemente grande */
            if (p->s.size == nunits)        /* exacto */
                prevp->s.ptr = p->s.ptr;
            else {                  /* asigna la parte final */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *) (p+1);
        }
        if (p == freep)    /* dio la vuelta a la lista libre */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* nada libre */
    }
}
```


La función **morecore** obtiene espacio de almacenamiento del sistema operativo. Los detalles de cómo lo hace varían de sistema a sistema. Debido a que pedir memoria al sistema es una operación comparativamente costosa, no deseamos hacerlo en cada llamada a **malloc**, así que **morecore** solicita al menos NALLOC unidades; este bloque grande será seccionado de acuerdo con las necesidades. Después de fijar el campo **size**, **morecore** inserta la memoria adicional llamando a **free**.

La llamada **sbrk(n)** al sistema UNIX regresa un apuntador a **n** bytes más de almacenamiento. **sbrk** regresa −1 si no hubo espacio, aunque NULL hubiera sido un mejor diseño. El −1 debe ser forzado a **char *** para que pueda ser comparado con el valor de retorno. Nuevamente, las conversiones forzadas hacen a la función relativamente inmune a los detalles de representación de apuntadores en máquinas diferentes. Hay, sin embargo, una suposición más; que los apuntadores a bloques diferentes regresados por **sbrk** pueden ser comparados. Esto no es garantizado por el estándar, que sólo permite la comparación de apuntadores dentro de un arreglo. Así, esta versión de **malloc** es portátil sólo entre máquinas para las que la comparación general de apuntadores es significativa.

```
#define NALLOC    1024    /* mínimo # de unidades por requerir */

/* morecore:  solicita más memoria al sistema */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)    /* no hay nada de espacio */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *) (up+1));
    return freep;
}
```

free es la última sección. Recorre la lista libre, iniciando en **freep**, buscando dónde insertar el bloque libre. Esto es entre dos bloques existentes o en uno de los extremos de la lista. En cualquier caso, si el bloque que está siendo liberado es adyacente a algún vecino, los bloques adyacentes se combinan. Los únicos problemas son mantener los apuntadores señalando a las cosas correctas y mantener los tamaños correctos.

```
/* free: coloca el bloque ap en la lista vacía */
void free(void * ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1;        /* apunta al encabezador de un bloque */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;    /* libera bloque al inicio o al final */

    if (bp + bp->s.size == p->s.ptr) { /* une al nbr superior */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {              /* une al nbr inferior */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Aunque la asignación de memoria es intrínsecamente dependiente de la máquina, el código anterior ilustra cómo pueden ser controladas las dependencias de la máquina y confinadas a una parte muy pequeña del programa. El uso de **typedef** y de union maneja la alineación (suponiendo que **sbrk** proporciona un apuntador apropiado). Las conversiones forzosas hacen que los apuntadores se manejen adecuada y explícitamente, e incluso se acoplan a una interfaz para el sistema mal diseñada. Aun cuando los detalles aquí están relacionados con la asignación de almacenamiento, el acercamiento general es aplicable también a otras situaciones.

Ejercicio 8-6. La función **calloc(n,size)** de la biblioteca estándar regresa un apuntador a **n** objetos de tamaño **size**, con el almacenamiento inicializado en cero. Escriba **calloc**, invocando a **malloc** o modificándola. □

Ejercicio 8-7. **malloc** acepta un tamaño solicitado sin verificar la posibilidad de que sea válido; **free** cree que el bloque que se pide liberar contiene un campo de tamaño correcto. Mejore esas rutinas para que se tomen más molestias en la revisión de errores. □

Ejercicio 8-8. Escriba una rutina **bfree(p,n)** que libere un bloque arbitrario **p** de **n** caracteres en la lista libre mantenida por **malloc** y **free**. Utilizando **bfree**, un usuario puede agregar un arreglo estático o externo a la lista libre en cualquier momento. □

APENDICE A: **Manual de referencia**

A1. Introducción

Este manual describe al lenguaje C tal como se especifica en *Draft Proposed American National Standard for Information Systems —Programming Language C,* documento número X3J11/88-001, con fecha 11 de enero de 1988. Este borrador no es el estándar final, y todavía es posible que ocurran algunos cambios en el lenguaje. Así pues, este manual no describe la definición final del lenguaje. Más aún es una interpretación del borrador propuesto del estándar, no el estándar en sí, aunque se ha tenido cuidado de hacerlo una guía confiable.

En su mayor parte, este manual sigue la línea amplia del borrador estándar, que a su vez sigue la de la primera edición de este libro, aunque la organización difiere en el detalle. Excepto por renombrar algunas producciones y porque no se formalizan las definiciones de los componentes léxicos o del preprocesador, la gramática dada aquí para el lenguaje es equivalente a la del borrador actual.

> En este manual, el material comentado se encuentra sangrado y escrito en un tipo más pequeño, como éste. A menudo estos comentarios resaltan las formas en las que el estándar ANSI de C difiere del lenguaje definido por la primera edición de este libro, o de refinamientos introducidos posteriormente en varios compiladores.

A2. Convenciones léxicas

Un programa consiste en una o más *unidades de traducción* almacenadas en archivos. Es traducido en varias fases, que se describen en §A12. Las primeras fases hacen transformaciones léxicas de bajo nivel, ejecutan directivas introducidas con líneas que principian con el carácter #, y realizan macrodefiniciones y expansiones. Cuando el preprocesamiento de §A12 está completo, el programa se ha reducido a una secuencia de componentes léxicos.

A2.1 Componentes léxicos (*tokens*)

Existen seis clases de componentes léxicos: identificadores, palabras reservadas, constantes, cadenas literales, operadores y otros separadores. Los blancos, tabuladores horizontales y verticales, nueva línea, avance de forma y comentarios, como se describen adelante (en su conjunto, llamados "espacio en blanco") son ignorados, excepto los que

separan componentes. Se requiere de algún espacio en blanco para separar identificadores de otra manera adyacentes, palabras reservadas y constantes.

Si el flujo de entrada se ha separado en componentes hasta un carácter determinado, el siguiente componente es la cadena más larga de caracteres que puede constituir uno.

A2.2 Comentarios

Los caracteres /* inician un comentario, que termina con los caracteres */. Los comentarios no se anidan y no pueden estar dentro de cadenas o caracteres literales.

A2.3 Identificadores

Un identificador es una secuencia de letras y dígitos. El primer carácter debe ser una letra; el subguión _ cuenta como una letra. Las letras minúsculas y mayúsculas son diferentes. Los identificadores pueden tener cualquier longitud y, para identificadores internos, al menos los primeros 31 caracteres son significativos; algunas implantaciones pueden hacer que más caracteres sean significativos. Los identificadores internos incluyen los nombres de macros del preprocesador y todos los otros nombres que no tienen ligado externo (§A11.2). Los identificadores con ligado externo están más restringidos: las implantaciones pueden hacer que sólo sean significativos seis caracteres y pueden ignorar la distinción entre mayúsculas y minúsculas.

A2.4 Palabras reservadas

Los siguientes identificadores son palabras reservadas y no se pueden utilizar de otra manera:

```
auto       double   int        struct
break      else     long       switch
case       enum     register   typedef
char       extern   return     union
const      float    short      unsigned
continue   for      signed     void
default    goto     sizeof     volatile
do         if       static     while
```

Algunas implantaciones también reservan las palabras **fortran** y **asm**.

Las palabras **const**, **signed** y **volatile** son nuevas en el estándar ANSI; **enum** y **void** son nuevas desde la primera edición, pero en uso común; **entry**, antes reservada pero nunca usada, ya no está reservada. Dependiendo de las decisiones del comité X3J11, la palabra **noalias** también puede estar reservada.

A2.5 Constantes

Hay varias clases de constantes. Cada una tiene un tipo de dato; en §A4.2 se discuten los tipos básicos.

constante:
constante-entera
constante-de-carácter
constante-flotante
constante-enumeración

A2.5.1 Constantes enteras

Una constante entera que consiste en una secuencia de dígitos se toma como octal si principia con 0 (dígito cero), de otra manera es decimal. Las constantes octales no contienen los dígitos 8 ó 9. Una secuencia de dígitos precedida por 0x ó 0X (dígito cero) se toma como un entero hexadecimal. Los dígitos hexadecimales incluyen de la a o A hasta la f o F con valores 10 al 15.

Una constante entera puede tener la letra u o U como sufijo, lo cual especifica que es **unsigned**. También puede tener como sufijo la letra l o L para estipular que es **long**.

El tipo de una constante entera depende de su forma, valor y sufijo (véase §A4 para una discusión de tipos). Un decimal sin sufijo tiene el primero de estos tipos, en el que su valor pueda ser representado: **int**, **long int**, **unsigned long int**. Si es octal o hexadecimal sin sufijo, tiene el primer valor posible de estos tipos: **int**, **unsigned int**, **long int**, **unsigned long int**. Si tiene el sufijo u o U, entonces es **unsigned int unsigned long int**. Si tiene el sufijo l o L, entonces es **long int unsigned long int**.

La elaboración de los tipos de constantes enteras va considerablemente más allá de la primera edición, que simplemente hacía que las grandes constantes enteras fueran **long**. Los sufijos U son nuevos.

A2.5.2 Constantes de carácter

Una constante de carácter es una secuencia de uno o más caracteres encerrados entre apóstrofos, como 'x'. El valor de una constante de carácter con un solo carácter es el valor numérico del carácter en el conjunto de caracteres de la máquina al tiempo de ejecución. El valor de una constante multicarácter está definido por la implantación.

Las constantes de carácter no contienen el carácter ' o nueva línea; para representarlos, así como a algunos otros caracteres, se pueden utilizar las siguientes secuencias de escape.

nueva línea	NL (LF)	\n	diagonal inversa	\	\\
tab horizontal	HT	\t	interrogación	?	\?
tab vertical	VT	\v	apóstrofo	'	\'
retroceso	BS	\b	comillas	"	\"
regreso de carro	CR	\r	número octal	*ooo*	*ooo*
avance de forma	FF	\f	número hexadecimal	*hh*	\x*hh*
señal audible	BEL	\a			

El escape *ooo* consiste en la diagonal inversa seguida por 1, 2 ó 3 dígitos octales, que estipulan el valor del carácter deseado. Un ejemplo común de esta construcción es \0 (no seguido por un dígito), que especifica el carácter NUL. El escape \x*hh* consiste en la diagonal inversa seguida por **x**, seguida por dígitos hexadecimales, que estipulan el valor de carácter deseado. No hay límite en el número de dígitos, pero el comportamiento queda indefinido si el valor de carácter resultante excede al del carácter más grande. Para caracteres octales

o hexadecimales, si la implantación trata al tipo **char** como **signed**, el valor es extendido en signo como si se forzara a ser de tipo **char**. Si el carácter que sigue a \ no es uno de los especificados, el comportamiento no está definido.

En algunas implantaciones, existe un conjunto extendido de caracteres que no se puede representar por el tipo **char**. Una constante en este conjunto extendido se escribe con una **L** precedente, por ejemplo **L'x'**, y se llama una constante de carácter amplio. Tal constante tiene tipo **wchar_t**, un tipo entero definido en el *header* **<stddef.h>**. Como con las constantes de carácter ordinarias, se pueden emplear escapes octales o hexadecimales; el efecto está indefinido si el valor especificado excede al que se representa con **wchar_t**.

Algunas de estas secuencias de escape son nuevas, en particular la representación hexadecimal de caracteres. Los caracteres extendidos también son nuevos. Los juegos de caracteres comúnmente usados en América y Europa occidental se pueden codificar para quedar en el tipo **char**; la intención principal de agregar **wchar_t** fue adaptarse a los lenguajes asiáticos.

A2.5.3 Constantes flotantes

Una constante flotante consta de una parte entera, un punto decimal, una parte fraccionaria, una **e** o **E**, un exponente entero signado optativo y un tipo sufijo optativo entre **f** o **F**, **l** o **L**. Las partes entera y fraccionaria constan de una secuencia de dígitos. Cualquiera de las partes entera o fraccionaria (no ambas) puede omitirse; cualquiera de las partes del punto decimal o la **e** y el exponente (no ambas) pueden omitirse. El tipo está determinado por el sufijo; **F** o **f** la hacen **float**, **L** o **l** la hacen **long double**; de otra manera es **double**.

Los sufijos en constantes flotantes son nuevos.

A2.5.4 Constantes de enumeración

Los identificadores declarados como enumeradores (véase §A8.4) son constantes de tipo **int**.

A2.6 Cadenas literales

Una cadena literal, también llamada cadena constante es una secuencia de caracteres delimitados por comillas, como en **"..."**. Una cadena tiene el tipo "arreglo de caracteres" y categoría de almacenamiento **static** (véase §A4, abajo) y se inicializa con los caracteres dados. El que cadenas idénticas sean distintas está definido por la implantación, y el comportamiento de un programa que intenta alterar una cadena literal está indefinido.

Cadenas literales adyacentes se concatenan en una sola cadena. Después de cualquier concatenación, se agrega un byte nulo **\0** a la cadena, de modo que los programas que rastrean la cadena puedan encontrar el fin. Las cadenas literales no contienen caracteres nueva línea o comillas; para representarlos, se usan las mismas secuencias de escape que para las constantes de carácter.

Como con las constantes de carácter, las cadenas literales en un conjunto de caracteres extendido se escriben con una **L** precedente, como en **L"..."**. Las cadenas literales amplias de caracteres tienen tipo "arreglo de **wchar_t**". La concatenación de cadenas literales ordinarias y amplias está indefinida.

La especificación de que las cadenas literales no tienen por qué ser distintas, y la prohibición en contra de modificarlas, son novedades dentro del estándar ANSI, así como la concatenación de cadenas literales adyacentes. Las cadenas literales de caracteres amplios son nuevas.

A3. Notación sintáctica

Dentro de la notación sintáctica que se emplea en este manual, las categorías sintácticas se indican con estilo *itálico*, y las palabras textuales y caracteres en estilo **mecanográfico**. Las categorías alternativas usualmente se listan en líneas separadas; en algunos casos, un conjunto amplio de alternativas cortas se presenta en una línea, marcada por la frase "uno de". Un símbolo optativo terminal o no terminal lleva el subíndice "*opt*", de modo que, por ejemplo,

$$\{ \; expresión_{opt} \; \}$$

significa una expresión optativa, encerrada entre llaves. La sintaxis se resume en §A13.

A diferencia de la gramática empleada en la primera edición de este libro, la que aquí se da hace explícita la precedencia y asociatividad de los operadores de expresión.

A4. Significado de los identificadores

Los identificadores, o nombres, se refieren a una variedad de cosas: funciones; rótulos de estructuras, uniones y enumeraciones; miembros de estructuras o de uniones; constantes de enumeración; nombres **typedef** y objetos. Un objeto, algunas veces llamado variable, es una localidad en el espacio de almacenamiento y su interpretación depende de dos atributos fundamentales: su *categoría de almacenamiento* y su *tipo*. La categoría de almacenamiento determina el tiempo de vida del almacenamiento asociado con el objeto identificado; el tipo determina el significado de los valores encontrados en el objeto identificado. Un nombre también tiene un alcance, que es la región del programa dentro de la que se conoce, y una liga, que determina si el mismo nombre en otro alcance se refiere al mismo objeto o función. El alcance y la liga se discuten en §A11.

A4.1 Categorías de almacenamiento

Existen dos categorías de almacenamiento: automática y estática. Varias palabras reservadas, junto con el contexto de la declaración de un objeto, especifican su categoría de almacenamiento. Los objetos automáticos son locales a un bloque (§A9.3), y son descartados al salir del bloque. Las declaraciones dentro de un bloque crean objetos automáticos si no se menciona una especificación de categoría de almacenamiento, o si se emplea el especificador **auto**. Los objetos declarados como **register** son automáticos, y se almacenan (si es posible) en registros rápidos de la máquina.

Los objetos estáticos pueden ser locales a un bloque o externos a todos los bloques, pero en cualquier caso mantienen su valor entre las salidas y reentradas a funciones o bloques. Dentro de un bloque, incluyendo uno que proporcione el código de una función, los objetos estáticos se declaran con la palabra reservada **static**. Los objetos que se decla-

ran fuera de todos los bloques, al mismo nivel que la definición de las funciones, son siempre estáticos. Se pueden hacer locales a una unidad de traducción en particular por el uso de la palabra reservada **static**; esto les otorga *liga interna*. Se hacen globales a un programa completo omitiendo una categoría explícita de almacenamiento, o utilizando la palabra reservada *extern*; esto les otorga *liga externa*.

A4.2 Tipos básicos

Existen varios tipos básicos. El *header* estándar **<limits.h>** que se describe en el apéndice B define los valores mayores y menores de cada tipo dentro de la implantación local. Los números dados en el apéndice B muestran las menores magnitudes aceptables.

Los objetos declarados como caracteres (**char**) son suficientemente grandes para almacenar cualquier miembro del conjunto de caracteres en ejecución. Si un carácter genuino de ese conjunto se almacena en un objeto **char**, su valor es equivalente al código entero para ese carácter, y es no negativo. Se pueden almacenar otras cantidades en variables **char**, pero el rango de valores disponibles, y en especial si el valor tiene signo, depende de la implantación.

Los caracteres sin signo declarados **unsigned char** consumen la misma cantidad de espacio que los caracteres sencillos, pero siempre aparecen como no negativos; los caracteres explícitamente signados que se declaran **signed char** toman igualmente el mismo espacio que los caracteres sencillos.

> **unsigned char** no aparece en la primera edición de este libro, pero es de uso común. **signed char** es nuevo.

Además de los tipos **char**, hay tres tamaños de enteros, declarados como **short int**, **int**, y **long int**. Los objetos **int** simples tienen el tamaño natural sugerido por la arquitectura de la máquina donde se ejecuta; los otros tamaños se proporcionan para cumplir con necesidades especiales. Los enteros más grandes proporcionan por lo menos tanto almacenamiento como los menores, pero la implantación puede hacer a los enteros simples equivalentes a los enteros cortos, o a los enteros largos. Todos los tipos **int** representan valores con signo a menos que se especifique lo contrario.

Los enteros sin signo, declarados mediante la palabra reservada **unsigned**, obedecen a las leyes de la aritmética módulo 2^n donde n es el número de bits en la representación, por lo que la aritmética sobre cantidades signadas nunca puede desbordarse. El conjunto de valores no negativos que se pueden almacenar en objetos con signo es un subconjunto de los que se pueden almacenar en el correspondiente objeto sin signo, y la representación para los valores en común es la misma.

Cualquiera de los tipos punto flotante de precisión sencilla (**float**), punto flotante de precisión doble (**double**) y punto flotante de precisión extra (**long double**) puede ser sinónimo, pero los últimos en la lista son al menos tan precisos como los que los anteceden.

> **long double** es nuevo. La primera edición hizo a **long float** equivalente a **double**; esto se ha rechazado.

Las enumeraciones son tipos únicos que tienen valores enteros; asociado con cada enumeración hay un conjunto de constantes nombradas (§A8.4). Las enumeraciones se comportan como enteros, pero es común que un compilador dé una advertencia cuando un objeto de un tipo de enumeración en particular se asigna a algo que no sea una de sus constantes o una expresión de su tipo.

Debido a que los objetos de estos tipos se pueden interpretar como números, se hará referencia a ellos como tipos *aritméticos*. Los tipos **char** e **int** de todos los tamaños, cada uno con o sin signo, y también los tipos de enumeración, se llamarán conjuntamente tipos *enteros*. Los tipos **float**, **double** y **long double** se llamarán tipos *flotantes*.

El tipo **void** especifica un conjunto vacío de valores. Se usa como el tipo regresado por funciones que no generan un valor.

A4.3 Tipos derivados

Además de los tipos básicos, existe una categoría conceptualmente infinita de tipos derivados, construidos a partir de los tipos fundamentales en las formas siguientes:

arreglos de objetos de un tipo dado;
funciones que regresan objetos de un tipo dado;
apuntadores a objetos de un tipo dado;
estructuras que contienen una secuencia de objetos de varios tipos;
uniones capaces de contener un objeto cualquiera de varios tipos.

En general, estos métodos de construcción de objetos se pueden aplicar en forma recursiva.

A4.4 Calificadores de tipo

Un tipo de objeto puede tener calificadores adicionales. El declarar **const** a un objeto anuncia que su valor no cambiará; declararlo **volatile** anuncia que tiene propiedades especiales de importancia para la optimización. Ningún calificador afecta el rango de valores o propiedades aritméticas del objeto. Los calificadores se discuten en §A8.2.

A.5 Objetos y valores-l

Un *objeto* es una región de almacenamiento con nombre; un *valor–l* es una expresión que se refiere a un objeto. Un ejemplo obvio de una expresión valor–l es un identificador con un tipo adecuado y una categoría de almacenamiento. Existen operadores que producen valores–l: por ejemplo, si **E** es una expresión de tipo apuntador, entonces ***E** es una expresión valor–l que se refiere al objeto al cual apunta **E**. El nombre "valor–l" proviene de la expresión de asignación **E1 = E2** en la que el operador izquierdo **E1** debe ser una expresión valor–l. La discusión de cada operador especifica si espera operandos valor–l y si entrega un valor–l.

A6. Conversiones

Algunos operadores pueden, dependiendo de sus operandos, provocar la conversión del valor de un operando de un tipo a otro. Esta sección explica el resultado que se espera de tales conversiones. §A6.5 resume las conversiones demandadas por la mayoría de los operadores ordinarios; en donde se requiera, será complementada con la discusión de cada operador.

A6.1 Promoción entera

Un carácter, un entero corto o un campo entero de bits, todos con o sin signo, o un objeto de tipo enumeración, se puede utilizar dentro de una expresión en cualquier lugar en donde se pueda usar un entero. Si un int puede representar a todos los valores del tipo original, entonces el valor es convertido a int; de otra manera el valor es convertido a unsigned int. Este proceso se llama *promoción entera*.

A6.2 Conversiones enteras

Un entero se convierte a un tipo sin signo dado encontrando el menor valor no negativo que sea congruente con ese entero, módulo uno más que el mayor valor que se pueda representar en el tipo sin signo. En una representación complemento a dos, esto es equivalente al truncamiento por la izquierda si el patrón de bits del tipo sin signo es más estrecho, y al llenado con ceros de valores sin signo y extensión de signo en valores con signo si el tipo sin signo es más amplio.

Cuando cualquier entero se convierte a un tipo con signo, el valor no se cambia si puede ser representado en el nuevo tipo, y en otro caso está definido por la implantación.

A6.3 Entero y flotante

Cuando un valor de tipo flotante se convierte a un tipo entero, la parte fraccionaria se descarta; si el valor resultante no puede ser representado en el tipo entero, el comportamiento no está definido. En particular, el resultado de convertir valores flotantes negativos a tipos enteros sin signo no está especificado.

Cuando un valor de tipo entero se convierte a flotante, y el valor está en el rango representable pero no es exactamente representable, entonces el resultado puede ser el valor siguiente más alto o más bajo. Si el resultado está fuera de rango, el comportamiento está indefinido.

A6.4 Tipos flotantes

Cuando un valor flotante menos preciso se convierte a un tipo flotante igual o más preciso, el valor no se modifica. Cuando un valor flotante más preciso se convierte a un tipo flotante menos preciso, y el valor está dentro del rango representable, el resultado puede ser el siguiente valor representable más alto o el siguiente más bajo. Si el resultado está fuera de rango, el comportamiento está indefinido.

A6.5 Conversiones aritméticas

Muchos operadores provocan una conversión y producen tipos resultantes en forma semejante. El efecto es pasar los operandos a un tipo común, que es también el tipo del resultado. A este patrón se le llama *conversiones aritméticas usuales*.

Primero, si cualquier operando es un **long double**, el otro es convertido a **long double**.
De otra manera, si cualquier operando es **double**, el otro es convertido a **double**.
De otra manera, si cualquier operando es **float**, el otro es convertido a **float**.
De otra manera, se realiza promoción entera en ambos operandos; después, si cualquier operando es **unsigned long int**, el otro es convertido a **unsigned long int**.

De otra manera, si un operando es **long int** y el otro es **unsigned int**, el efecto depende de si un **long int** puede representar a todos los valores de un **unsigned int**; si es así, el operando **unsigned int** es convertido a **long int**; si no lo es, ambos son convertidos a **unsigned long int**.
De otra manera, si un operando es **long int**, el otro es convertido a **long int**.
De otra manera, si cualquier operando es **unsigned int**, el otro es convertido a **unsigned int**.
De otra manera, ambos operandos tienen tipo **int**.

> Aquí hay dos cambios. Primero, la aritmética sobre operandos float se puede realizar en precisión sencilla, en lugar de doble; la primera edición especificaba que toda la aritmética flotante era de doble precisión. Segundo, los tipos sin signo más pequeños, cuando se combinan con un tipo con signo mayor, no propagan la propiedad de no signado al tipo resultante; en la primera edición, siempre dominaba lo no signado. Las nuevas reglas son ligeramente más complicadas, pero de alguna forma reducen las sorpresas que pueden ocurrir cuando una cantidad sin signo encuentra a otra con signo. Aún pueden ocurrir resultados inesperados cuando una expresión sin signo es comparada con una expresión con signo del mismo tamaño.

A6.6 Apuntadores y enteros

Una expresión de tipo entero puede ser sumada o restada de un apuntador; en tal caso, la expresión entera es convertida tal como se especifica en la discusión del operador de adición (§A7.7).

Dos apuntadores a objetos del mismo tipo, dentro del mismo arreglo, pueden ser restados; el resultado es convertido a un entero como se especifica en la discusión del operador de sustracción (§A7.7).

Una expresión entera constante con valor 0, o esa expresión forzada al tipo **void ***, puede ser convertida, por medio de un *cast*, por asignación, o por comparación, a un apuntador de cualquier tipo. Esto produce un apuntador nulo que es igual a otro apuntador nulo del mismo tipo, pero diferente a cualquier apuntador a una función u objeto.

Se permiten otras ciertas conversiones que involucran apuntadores, pero tienen aspectos dependientes de la implantación. Se deben especificar con un operador explícito de conversión de tipo o cast (§§A7.5 y A8.8).

Un apuntador se puede convertir a un tipo entero suficientemente grande para mantenerlo; el tamaño requerido depende de la implantación. La función de mapeo también depende de la implantación.

Un objeto de tipo entero se puede explícitamente convertir a un apuntador. El mapeo siempre lleva un entero suficientemente amplio convertido de un apuntador de regreso al mismo apuntador, pero de otra manera es dependiente de la implantación.

Un apuntador de un tipo se puede convertir a un apuntador a otro tipo. El apuntador resultante puede causar errores de direccionamiento si no se refiere a un objeto adecuadamente alineado en la memoria. Se garantiza que un apuntador a un objeto se puede convertir a un apuntador a un objeto cuyo tipo requiere de una menor o igualmente estricta alineación en el almacenamiento y regresado de nuevo sin cambio; la noción de "alineación" es dependiente de la implantación, pero los objetos de tipo **char** tienen los requisitos de alineación menos estrictos. Como se describe en §A6.8, un apuntador se puede convertir a tipo **void *** y regresado de nuevo sin cambio.

Finalmente, un apuntador a una función se puede convertir a un apuntador a otro tipo de función. La llamada a la función especificada por el apuntador convertido es depen-

diente de la implantación; sin embargo, si el apuntador convertido es reconvertido a su tipo original, el resultado es idéntico al apuntador original.

A6.7 Void

El (inexistente) valor de un objeto void no se puede utilizar en ninguna forma, ni se puede aplicar la conversión explícita o implícita a ningún tipo no void. Debido a que la expresión void denota un valor inexistente, sólo se puede utilizar donde no sea requerido el valor, por ejemplo, una proposición de expresión (§A9.2) o el operando izquierdo de un operador coma (§A7.18).

Una expresión se puede convertir a tipo void con un *cast*. Por ejemplo, una conversión forzada a void deja documentado el rechazo del valor de una llamada a función utilizada como proposición de expresión.

> void no aparecía en la primera edición de este libro, pero se ha vuelto común desde entonces.

A6.8 Apuntadores a void

Cualquier apuntador se puede convertir a tipo void * sin pérdida de información. Si el resultado se regresa al tipo de apuntador original, éste es recuperado. A diferencia de la conversión apuntador-a-apuntador discutida en §A6.6, que requiere un *cast* explícito, los apuntadores pueden ser asignados hacia y desde apuntadores de tipo void * y pueden ser comparados con ellos.

> Esta interpretación de apuntadores void * es nueva; anteriormente, los apuntadores char * jugaban el papel de apuntadores genéricos. El estándar ANSI específicamente consiente el encuentro de apuntadores void * con apuntadores a objetos en asignaciones y relaciones, mientras que requiere *cast* explícitos para otras mezclas de apuntadores.

A7. Expresiones

La precedencia de los operadores en expresiones es la misma que el orden de las subsecciones principales de esta sección, primero la más alta precedencia. Así, por ejemplo, las expresiones a las que se hace referencia como operandos de + (§A7.7) son las definidas en §§A7.1-A7.6. Dentro de cada subsección, los operadores tienen la misma precedencia. En cada subsección se especifica la asociatividad por la izquierda o la derecha para los operadores discutidos allí. La gramática incorpora la precedencia y asociatividad de los operadores de la expresión y se resume en §A13.

La precedencia y asociatividad de los operadores está especificada completamente, pero el orden de evaluación de las expresiones está, con ciertas excepciones, indefinido, aún si las subexpresiones involucran efectos colaterales. Esto es, a menos que la definición de un operador garantice que sus operandos se evalúen en un orden particular, la implantación está en libertad de evaluar los operandos en cualquier orden, o incluso intercalar su evaluación. Sin embargo, cada operador combina los valores producidos por sus operandos en una forma compatible con el análisis gramatical de la expresión en que aparece.

> El comité ANSI decidió, últimamente en sus reportes, restringir la anterior libertad de reordenar las expresiones que involucran operadores matemáticamente conmutativos y asociativos, pero que pueden no ser asociativos computacionalmente. En la práctica, el cambio sólo afecta a los cálculos de punto flotante cercanos a los límites de su precisión y en situaciones en donde es posible el desbordamiento.

El manejo del desbordamiento, errores de división y otras condiciones de error dentro de la evaluación de expresiones no está definido por el lenguaje. La mayoría de las realizaciones existentes de C ignoran el desbordamiento en la evaluación de expresiones y asignaciones enteras con signo, pero este comportamiento no está garantizado. El trato de la división entre 0, y todas las condiciones de error de punto flotante, varía entre las implantaciones; algunas veces es ajustable mediante el uso de funciones no estándar de biblioteca.

A7.1 Generación de apuntadores

Si el tipo de una expresión o subexpresión es un "arreglo de *T*", para algún tipo *T*, entonces el valor de la expresión es un apuntador al primer objeto del arreglo, y el tipo de la expresión es alterado a "apuntador a *T*". Esta conversión no sucede si la expresión es el operando de un operador & unario, o de ++, --, **sizeof**, o el operando izquierdo de un operador de asignación o "el operador ." De modo semejante, una expresión de tipo "función que regresa *T*", excepto cuando se utiliza como el operando del operador &, es convertida a "apuntador a función que regresa *T*". Una expresión que ha sufrido una de estas conversiones no es valor—l.

A7.2 Expresiones primarias

Las expresiones primarias son identificadores, constantes, cadenas, o expresiones entre paréntesis.

expresión primaria:
identificador
constante
cadena
(*expresión*)

Un identificador es una expresión primaria, siempre que haya sido declarado adecuadamente tal como se discutió anteriormente. Su tipo está especificado por su declaración. Un identificador es un valor—l si se refiere a un objeto (§A5) y si su tipo es aritmético, estructura, unión o apuntador.

Una constante es una expresión primaria. Su tipo depende de su forma, tal como se discutió en §A2.5.

Una cadena es una expresión primaria. Su tipo es originalmente "arreglo de char" (para cadenas de caracteres amplios, "arreglo de wchar_t"), pero siguiendo la regla dada en §A7.1, usualmente se modifica a "apuntador a char" (wchar_t) y el resultado es un apuntador al primer carácter de la cadena. La conversión tampoco ocurre en ciertos inicializadores; véase §A8.7.

Una expresión entre paréntesis es una expresión primaria cuyo tipo y valor son idénticos a los de una expresión que no lo esté. La presencia de paréntesis no afecta el que la expresión sea un valor—l.

A7.3 Expresiones posfijas

Los operadores de expresiones posfijas se agrupan de izquierda a derecha.

expresión-posfija:
    expresión-primaria
    expresión-posfija [*expresión*]
    expresión-posfija (*lista-de-expresiones-argumento*$_{opt}$)
    expresión-posfija . *identificador*
    expresión-posfija -> *identificador*
    expresión-posfija ++
    expresión-posfija --

lista-expresiones-argumento:
    expresión-de-asignación
    lista-expresiones-argumento , *expresión-de-asignación*

A7.3.1 Referencias a arreglos

Una expresión posfija seguida por una expresión dentro de corchetes es una expresión posfija que denota una referencia indexada a un arreglo. Una de las dos expresiones debe tener tipo "apuntador a *T*", donde *T* es algún tipo, y la otra debe tener tipo entero; el tipo de la expresión subíndice es *T*. La expresión **E1[E2]** es idéntica (por definición) a ***((E1) + (E2))**. Véase §A8.6.2 para una discusión adicional.

A7.3.2 Llamadas a funciones

Una llamada a función es una expresión posfija, conocida como designador de función, seguido de paréntesis que contienen una lista posiblemente vacía de expresiones de asignación separadas por comas (§A7.17), que constituyen los argumentos de la función. Si la expresión posfija consiste en un identificador para el que no existe una declaración dentro del alcance actual, el identificador es explícitamente declarado como si la declaración

extern int *identificador* ( ) ;

hubiese sido dada en el bloque más interno que contenga la llamada a la función. La expresión posfija (después de una posible declaración implícita y generación de apuntador, §A7.1) debe ser del tipo "apuntador a función que regresa *T*", para algún tipo de *T*, y el valor de la llamada a la función tiene el tipo *T*.

> En la primera edición, el tipo estaba restringido a "función" y se requería de un operador ***** explícito para invocar a través de apuntadores a funciones. El estándar ANSI está de acuerdo con algunos compiladores existentes permitiendo la misma sintaxis para llamadas a funciones y a funciones especificadas por apuntadores. La sintaxis anterior aún se puede utilizar.

El término *argumento* se utiliza para una expresión pasada por una llamada a función; el término *parámetro* se emplea para un objeto de entrada (o su identificador) recibido por una definición de función o descrito dentro de la declaración de una función. Los términos "argumento (parámetro real)" y "argumento (parámetro) formal" respectivamente, se usan algunas veces para hacer la misma distinción.

En preparación para la llamada a una función, se hace una copia de cada argumento; todo el paso de argumentos es estrictamente por valor. Una función puede cambiar los valores de sus objetos parámetros, que son copias de las expresiones argumentos, pero estos cambios no pueden afectar los valores de los argumentos. Sin embargo, es posible pasar un apuntador en el entendimiento de que la función puede cambiar el valor del objeto al que apunta el apuntador.

Existen dos estilos en los que se pueden declarar las funciones. En el nuevo estilo, los tipos de los parámetros son explícitos y son parte del tipo de la función; tal declaración se llama también el prototipo de la función. En el estilo anterior, los tipos de los parámetros no se especifican. La declaración de una función se trata en §§A8.6.3 y A10.1.

Si la declaración de función dentro del alcance de una llamada está en el estilo anterior, entonces la promoción de argumentos predefinida se aplica en cada argumento como sigue: la promoción entera (§A6.1) se realiza en cada argumento de tipo entero, y cada argumento **float** es convertido a **double**. El efecto de la llamada queda indefinido si el número de argumentos no coincide con el número de parámetros de la definición de la función, o si el tipo de un argumento después de la promoción no coincide con el del parámetro correspondiente. La coincidencia de tipos depende de si la función de la función está en el nuevo o el viejo estilo. Si está en el anterior, entonces la comparación es entre el tipo promovido del argumento de la llamada y el tipo promovido del parámetro; si la definición está en el estilo nuevo, el tipo promovido del argumento debe ser el del parámetro en sí, sin promoción.

Si la declaración de la función en el alcance de una llamada está en estilo nuevo, entonces los argumentos se convierten, como por asignación, a los tipos de los parámetros correspondientes del prototipo de la función. El número de argumentos debe ser el mismo que el número de parámetros explícitamente declarados, a menos de que la lista de parámetros de la declaración termine con la notación de coma y tres puntos (, ...). En ese caso, el número de argumentos debe igualar o exceder al número de parámetros; los argumentos más allá de los parámetros con tipo explícitamente declarado sufren la promoción predefinida de argumentos descrita en el párrafo precedente. Si la definición de la función está en el estilo anterior, entonces el tipo de cada parámetro dentro del prototipo visible a la llamada debe coincidir con los parámetros correspondientes de la definición, después de que al tipo de parámetro de la definición se le ha hecho la promoción de argumentos.

> Estas reglas son especialmente complicadas debido a que deben satisfacer una mezcla de funciones en el nuevo y viejo estilos. Las mezclas se deben evitar si es posible.

El orden de evaluación de los argumentos no está especificado; nótese que los compiladores difieren. Sin embargo, los argumentos y los designadores de función son completamente evaluados, incluyendo todos los efectos colaterales, antes de que se entre a la función. Se permiten las llamadas recursivas a cualquier función.

A7.3.3 Referencias a estructuras

Una expresión posfija seguida por un punto seguido de un identificador es una expresión posfija. El primer operando de la expresión debe ser una estructura o una unión, y

el identificador debe nombrar a un miembro de la estructura o unión. El valor es el miembro nombrado de la estructura o unión y su tipo es el tipo del miembro. La expresión es un valor—l si la primera expresión es un valor—l, y si el tipo de la segunda expresión no es un tipo arreglo.

Una expresión posfija seguida por una flecha (construida con – y >) seguida por un identificador es una expresión posfija. El primer operando en la expresión debe ser un apuntador a una estructura o una unión y el identificador debe nombrar a un miembro de la estructura o unión. El resultado se refiere al miembro nombrado de la estructura o unión al cual apunta el apuntador de la expresión, y el tipo es el tipo del miembro; el resultado es un valor—l si el tipo no es arreglo.

Así la expresión E1->MOS es lo mismo que (*E1).MOS. Las estructuras y uniones se discuten en §A8.3.

En la primera edición de este libro ya estaba la regla de que un nombre de miembro en una expresión así tenía que pertenecer a la estructura o unión mencionada en la expresión posfija; sin embargo, una nota admitía que esta regla no se seguía firmemente. Los compiladores recientes y el ANSI la siguen.

A7.3.4 Incrementos posfijos

Una expresión posfija seguida de un operador ++ o -- es una expresión posfija. El valor de la expresión es el valor del operando. Después de usar el valor, se incrementa el operando (++) o se decrementa (--) en 1. El operando debe ser un valor—l; véase las discusión de operadores aditivos (§A.7.7) y de asignación (§A.7.17) para posteriores restricciones en el operando y detalles de la operación. El resultado no es un valor—l.

A7.4 Operadores unarios

Las expresiones con operadores unarios se agrupan de derecha a izquierda.

expresión-unaria:
  expresión-posfija
  ++ *expresión-unaria*
  -- *expresión-unaria*
  operador-unario expresión-cast
  sizeof *expresión-unaria*
  sizeof (*nombre-de-tipo*)
operador-unario: uno de
  & * + - ~ !

A7.4.1 Operadores prefijos de incremento

Una expresión unaria precedida por un operador ++ o -- es una expresión unaria. El operando se incrementa (++) o decrementa (--) en 1. El valor de la expresión es el valor después del incremento (decremento). El operando debe ser un valor–l; véase la discusión de operadores aditivos (§A7.7) y de asignación (§A7.17) para posteriores restricciones en el operando y detalles de la operación. El resultado no es un valor–l.

A7.4.2 Operador de dirección

El operador unario & toma la dirección de su operando. El operando debe ser el valor–l que no se refiera ni a un campo de bits ni a un objeto declarado como **register**, o debe ser de tipo función. El resultado es un apuntador al objeto o función al que se refiere el valor–l. Si el tipo del operando es *T*, el tipo del resultado es "apuntador a *T*".

A7.4.3 Operador de indirección

El operador unario * denota indirección y regresa el objeto o función a que apunta su operando. Es un valor–l si el operando es un apuntador a un objeto de tipo aritmético, estructura, unión o apuntador. Si el tipo de la expresión es un "apuntador a *T*", el tipo del resultado es *T*.

A7.4.4 Operador más unario

El operando del operador unario + debe tener tipo aritmético o apuntador y el resultado es el valor del operando. Un operando entero sufre promoción entera. El tipo del resultado es el tipo del operando promovido.

El + unario es nuevo en el estándar ANSI. Se agregó por simetría con el – unario.

A7.4.5 Operador menos unario

El operador del – unario debe tener tipo aritmético y el resultado es el negativo de su operando. Un operando entero sufre promoción entera. El negativo de una cantidad sin signo se calcula restando el valor promovido del mayor valor del tipo promovido y agregándole uno; pero el cero negativo es cero. El tipo del resultado es el tipo del operando promovido.

A7.4.6 Operador complemento a uno

El operando del operador unario ~ debe tener tipo entero y el resultado es el complemento a uno de su operando. Se realiza promoción entera. Si el operando es sin signo, el resultado se calcula restando el valor del mayor valor del tipo promovido. Si el operando es con signo, el resultado se calcula convirtiendo el operando promovido al tipo sin signo correspondiente, aplicando ~ y regresando al tipo con signo. El tipo del resultado es el tipo del operando promovido.

A7.4.7 Operador de negación lógica

El operando del operador ! debe tener tipo aritmético o ser un apuntador, y el resultado es 1 si el valor de su operando es compara igual a 0, y 0 en caso contrario. El tipo del resultado es **int**.

A7.4.8 Operador sizeof

El operador **sizeof** produce el número de bytes requeridos para almacenar un objeto del tipo de su operando. El operando es una expresión, que no es evaluada, o un nombre

de tipo entre paréntesis. Cuando **sizeof** se aplica a **char**, el resultado es 1; cuando se aplica a un arreglo, el resultado es el número total de bytes en el arreglo. Cuando se aplica a una estructura o unión, el resultado es el número de bytes en el objeto, incluyendo cualquier relleno requerido para completar a un arreglo: el tamaño de un arreglo de *n* elementos es *n* veces el tamaño de un elemento. El operador no se puede aplicar a un operando de tipo función o de tipo incompleto, o a un campo de bits. El resultado es un entero constante sin signo; el tipo particular se define por la implantación. El *header* estándar **<stddef.h>** (véase el apéndice B) define este tipo como **size_t**.

A7.5 Cast

Una expresión unaria precedida por el nombre entre paréntesis de un tipo provoca la conversión del valor de la expresión al tipo nombrado.

> *expresión-cast:*
> *expresión-unaria*
> (*nombre-de-tipo*) *expresión-cast*

Esta construcción se llama *cast* (conversión forzada). Los nombres de tipo se describen en §A8.8. Los efectos de conversión son descritos en §A6. Una expresión con un *cast* no es un valor—l.

A7.6 Operadores multiplicativos

Los operadores multiplicativos *, /, y % se agrupan de izquierda a derecha.

> *expresión-multiplicativa:*
> *expresión-cast*
> *expresión-multiplicativa * expresión-cast*
> *expresión-multiplicativa / expresión-cast*
> *expresión-multiplicativa % expresión-cast*

Los operandos de * y / deben tener tipo aritmético; los operandos de % deben tener tipo entero. Las conversiones aritméticas usuales se realizan sobre los operandos, y predicen el tipo del resultado.

El operador binario * denota multiplicación.

El operador binario / produce el cociente y el operador % el residuo de la división del primer operando entre el segundo; si el segundo operando es 0, el resultado está indefinido. De otra manera, siempre es cierto que (a/b) * b + a%b es igual que **a**. Si ninguno de los operandos es negativo, entonces el residuo es no negativo y menor que el divisor; si no lo son, se garantiza sólo que el valor absoluto del residuo es menor que el valor absoluto del divisor.

A7.7 Operadores aditivos

Los operadores aditivos + y − se agrupan de izquierda a derecha. Si los operandos tienen tipo aritmético, se realizan las conversiones aritméticas usuales. Existen algunas posibilidades adicionales de tipos para cada operador.

> *expresión-aditiva:*
> *expresión-multiplicativa*
> *expresión-aditiva + expresión-multiplicativa*
> *expresión-aditiva − expresión-multiplicativa*

El resultado del operador + es la suma de los operandos. Un apuntador a un objeto que esté en un arreglo y un valor de cualquier tipo entero se pueden sumar. Lo último se convierte a una dirección de desplazamiento, multiplicándolo por el tamaño del objeto al que el apuntador apunta. La suma es un apuntador del mismo tipo que el apuntador original y apunta a otro objeto dentro del mismo arreglo, desplazado apropiadamente del objeto original. Así, si **P** es un apuntador a un objeto en un arreglo, la expresión **P + 1** en un apuntador al siguiente objeto en el arreglo. Si el apuntador de la suma apunta fuera de los límites del arreglo, excepto a la primera localidad más allá del final, el resultado es indefinido.

La posibilidad de apuntadores más allá del final del arreglo es nueva. Esto legitimiza una expresión idiomática común para iterar sobre los elementos de un arreglo.

El resultado del operador − es la diferencia de los operandos. Un valor de cualquier tipo entero se puede restar de un apuntador, y se aplican las mismas conversiones y condiciones que para la adición.

Si se restan dos apuntadores a objetos del mismo tipo, el resultado es un valor entero con signo que representa el desplazamiento entre los objetos apuntados; los apuntadores a objetos sucesivos difieren en 1. El tipo del resultado depende de la implantación, pero está definido como **ptrdiff_t** en el *header* estándar **<stddef.h>**. El valor está indefinido a menos de que los apuntadores apunten a objetos dentro del mismo arreglo; sin embargo, si **P** apunta al último miembro de un arreglo, entonces **(P+1)−P** tiene valor 1.

A7.8 Operadores de corrimiento

Los operadores de corrimiento << y >> se agrupan de izquierda a derecha. Para ambos operadores, cada operando debe ser entero y está sujeto a las promociones enteras. El tipo del resultado es el del operando promovido de la izquierda. El resultado está indefinido si el operando de la derecha es negativo, mayor o igual al número de bits del tipo de la expresión de la izquierda.

> *expresión-de-corrimiento:*
> *expresión-aditiva*
> *expresión-de-corrimiento << expresión-aditiva*
> *expresión-de-corrimiento >> expresión-aditiva*

El valor de **E1<<E2** es **E1** (interpretado como un patrón de bits) recorrido a la izquierda **E2** bits; en ausencia de desbordamiento, esto es equivalente a la multiplicación por 2^{E2}. El valor de **E1>>E2** es **E1** recorrido a la derecha **E2** posiciones de bits. El corrimiento a la derecha es equivalente a la división entre 2^{E2} si **E1** es no tiene signo o si tiene un valor no negativo; de otra forma el resultado está definido por la implantación.

A7.9 Operadores de relación

Los operadores de relación se agrupan de izquierda a derecha, pero esto no es de utilidad, **a<b<c** se analiza como **(a<b)<c**, y **a<b** se evalúa como 0 o como 1.

expresión-relacional:
expresión-de-corrimiento
expresión-relacional < *expresión-de-corrimiento*
expresión-relacional > *expresión-de-corrimiento*
expresión-relacional <= *expresión-de-corrimiento*
expresión-relacional >= *expresión-de-corrimiento*

Los operadores < (menor que), > (mayor que), <= (menor o igual a) y >= (mayor o igual a) dan todos 0 si la relación especificada es falsa, y 1 si es verdadera. El tipo del resultado es **int**. Las conversiones aritméticas usuales se realizan sobre los operandos aritméticos. Pueden compararse los apuntadores a objetos del mismo tipo; el resultado depende de las localidades relativas en el espacio de direccionamiento de los objetos apuntados. La comparación de apuntadores está definida sólo para partes del mismo objeto: si dos apuntadores apuntan al mismo objeto simple, se comparan como iguales; si los apuntadores lo hacen a miembros de la misma estructura, los apuntadores a objetos declarados más adelante en la estructura se comparan como mayores; si los apuntadores son a miembros de la misma unión, se comparan como iguales; si los apuntadores hacen referencia a miembros de un arreglo, la comparación es equivalente a la comparación de los correspondientes subíndices. Si P apunta al último miembro de un arreglo, entonces P+1 se compara como mayor que P, incluso aunque P+1 apunte fuera del arreglo. De otra manera, la comparación de apuntadores está indefinida.

Estas reglas liberan algo las restricciones establecidas en la primera edición, permitiendo la comparación de apuntadores a diferentes miembros de una estructura o unión. También legalizan la comparación con un apuntador justo más allá del final de un arreglo.

A7.10 Operadores de igualdad

expresión-de-igualdad:
expresión-relacional
expresión-de-igualdad == *expresión-relacional*
expresión-de-igualdad != *expresión-relacional*

Los operadores == (igual a) y != (no igual a) son análogos a los operadores de relación excepto por su menor precedencia. (Así a<b == c<d es 1 cuando a<b y c<d tengan los mismos valores de verdad).

Los operadores de igualdad siguen las mismas reglas que los operadores de relación, pero permiten posibilidades adicionales: un apuntador puede ser comparado con una expresión constante entera con valor 0, o con un apuntador a **void**. Véase §A.6.6.

A7.11 Operador AND para bits

expresión-AND:
expresión-de-igualdad
expresión-AND & *expresión-de-igualdad*

Las conversiones aritméticas usuales se realizan; el resultado es la función AND de bits de los operandos. El operador se aplica sólo a operandos enteros.

A7.12 Operador OR exclusivo para bits

expresión-OR-exclusivo:
expresión-AND
expresión-OR-exclusivo ^ *expresión-AND*

Se realizan las conversiones aritméticas usuales; el resultado es la función OR exclusivo de los operandos. El operador se aplica sólo a operandos enteros.

A7.13 Operador OR inclusivo para bits

expresión-OR-inclusivo:
expresión-OR-exclusivo
expresión-OR-inclusivo | *expresión-OR-exclusivo*

Se realizan las conversiones aritméticas usuales; el resultado es la función OR inclusivo de sus operandos. El operador se aplica sólo a operandos enteros.

A7.14 Operador lógico AND

expresión-lógica-AND:
expresión-OR-inclusivo
expresión-lógica-AND && *expresión-OR-inclusivo*

El operador && se agrupa de izquierda a derecha. Regresa 1 si ambos operandos se comparan como diferentes de cero; de otra manera, regresa 0. A diferencia de &, && garantiza la evaluación de izquierda a derecha: el primer operando es evaluado, incluyendo todos los efectos colaterales; si es igual a 0, el valor de la expresión es 0. De otra manera, el operando derecho es evaluado, y si es igual a 0, el valor de la expresión es 0; de otra manera es 1.

Los operandos no requieren de tener el mismo tipo, pero cada uno debe tener tipo aritmético o ser un apuntador. El resultado es **int**.

A7.15 Operador lógico OR

expresión-lógica-OR:
expresión-lógica-AND
expresión-lógica-OR || *expresión-lógica-AND*

El operador || se agrupa de izquiera a derecha. Regresa 1 si alguno de sus operandos no se compara como cero, y 0 en caso contrario. A diferencia de |, || garantiza la evaluación de izquierda a derecha: el primer operando es evaluado, incluyendo los efectos colaterales; si es diferente de 0, el valor de la expresión es 1. De otra manera, es evaluado el operando derecho y, si es diferente de 0, el valor de la expresión es 1; de otra manera es cero.

Los operandos no requieren tener el mismo tipo, pero cada uno de ellos debe tener numérico o ser apuntador. El resultado es **int**.

A7.16 Operador condicional

expresión condicional:
  expresión-lógica-OR
  expresión-lógica-OR ? *expresión* : *expresión-condicional*

La primera expresión se evalúa, incluyendo todos los efectos colaterales; si se compara como diferente de 0, el resultado es el valor de la segunda expresión; de otra manera, es el de la tercera expresión. Sólo se evalúa uno de los operadores segundo o tercero. Si el segundo y el tercer operandos son aritméticos, se realizan las conversiones aritméticas usuales para hacerlos de algún tipo común y ese es el tipo del resultado. Si ambos son **void**, estructuras o uniones del mismo tipo, o apuntadores a objetos del mismo tipo, el resultado tiene el tipo común. Si uno es un apuntador y el otro la constante 0, el 0 es convertido a tipo apuntador y el resultado tiene ese tipo. Si uno es un apuntador a **void** y el otro es otro apuntador, el otro apuntador es convertido a apuntador a **void** y ése es el tipo del resultado.

En la comparación de tipos para apuntadores, los calificadores de tipo (§A8.2) en el tipo al que apunta el apuntador no importan, pero el resultado hereda los calificadores de ambas ramas de la condicional.

A7.17 Expresiones de asignación

Existen varios operadores de asignación; todos se agrupan de derecha a izquierda.

expresión-de-asignación:
  expresión-condicional
  expresión-unaria operador-de-asignación expresión-de-asignación

operador-de-asignación: uno de

```
= *= /= %= += -= <<= >>= &= ^= |=
```

Todos requieren de un valor—l como operando izquierdo y éste debe ser modificable: no debe ser un arreglo y no debe tener un tipo incompleto ni ser una función. Tampoco debe ser calificado con **const**; si es una estructura o unión, no debe tener ningún miembro o, recursivamente ningún submiembro calificado con **const**. El tipo de una expresión de asignación es el de su operando izquierdo, y el valor es el almacenado en el operando izquierdo después de que ha tenido lugar la asignación.

En la asignación simple con =, el valor de la expresión reemplaza al del objeto al que se hace referencia con el valor—l. Uno de los siguientes debe ser verdadero: ambos operandos tienen tipo aritmético, en tal caso, el operando de la derecha es convertido al tipo del operando izquierdo por la asignación; o ambos operandos son estructuras o uniones del mismo tipo; o un operando es un apuntador y el otro es un apuntador a void; o el operando izquierdo es un apuntador y el operando derecho es una expresión constante con valor 0; o ambos operandos son apuntadores a funciones u objetos cuyos tipos son los mismos excepto por la posible ausencia de **const** o **volatile** en el operando derecho.

Una expresión de la forma E1 *op* = E2 es equivalente a E1 = E1 *op* (E2) excepto que E1 es evaluado sólo una vez.

De acuerdo con las restricciones anteriores, es ilegal asignar apuntadores cuando el lado derecho apunta a un objeto de algún tipo y el lado izquierdo apunta a un objeto con una versión calificada con **const** de ese tipo. Una lectura estricta de esta regla y su análoga para **cast** causa dificultades al implantar ciertas funciones de biblioteca; sería bueno que fuera menos restricta.

A7.18 Operador coma

expresión:
  expresión-de-asignación
  expresión , *expresión-de-asignación*

Un par de expresiones separadas por una coma se evalúa de izquierda a derecha y el valor de la expresión izquierda se descarta. El tipo y valor del resultado son el tipo y valor del operando de la derecha. Todos los efectos colaterales de la evaluación del operando izquierdo se completan antes de principiar la evaluación del operando derecho. En contextos donde a la coma se le da un significado especial, por ejemplo en listas de argumentos para funciones (§A7.3.2) y listas de inicializadores (§A8.7), la unidad sintáctica requerida es una expresión de asignación, de modo que el operador coma sólo aparece en una agrupación limitada por paréntesis; por ejemplo,

```
f(a, (t=3, t+2), c)
```

tiene tres argumentos, el segundo de los cuales tiene el valor 5.

A7.19 Expresiones constantes

Sintácticamente, una expresión constante es una expresión restringida a un subconjunto de operadores:

expresión-constante:
  expresión-condicional

Las expresiones que se evalúan a una constante se requieren en varios contextos: después de **case**, como límites de un arreglo y longitudes de campos de bits, como valor de una constante de enumeración, en inicializadores y dentro de ciertas expresiones del preprocesador.

Las expresiones constantes no pueden contener asignaciones, operadores de incremento o decremento, llamadas a funciones ni operadores coma, excepto en un operando de **sizeof**. Si se requiere que la expresión constante sea entera, sus operandos deben consistir en constantes enteras, de enumeración, de caracteres y flotantes; los **cast** deben estipular un tipo entero y cualquier constante flotante debe ser convertida a entero. Esto necesariamente excluye operaciones sobre arreglos, indirección, dirección-de y miembros de estructura. (Sin embargo, se permite cualquier operando para **sizeof**.)

Hay más libertad para las expresiones constantes de inicializadores; los operandos pueden ser de cualquier tipo de constante y el operando unario & puede ser aplicado a objetos externos o estáticos y a arreglos externos o estáticos, indexados con una expresión constante. El operador unario & también se puede aplicar implícitamente por la aparición de arreglos no indexados y funciones. Los inicializadores deben evaluarse a una constante o a la dirección de un objeto externo o estático previamente declarado más o menos una constante.

Se permite menos libertad para las expresiones enteras constantes después de **#if**; no se permiten expresiones **sizeof**, constantes de enumeración ni cast. Véase §A12.5.

A8. Declaraciones

Las declaraciones especifican la interpretación dada a cada identificador; no necesariamente reservan espacio de almacenamiento asociado con el identificador. Las declaraciones que reservan almacenamiento se llaman *definiciones*. Las declaraciones tienen la forma

declaración:
especificadores-de-declaración lista-de-declaradores-init$_{opt}$;

Los declaradores que están en la lista-de-declaradores-init contienen la lista de identificadores que están siendo declarados; los especificadores-de-declaración consisten en una secuencia de especificadores de tipo y categoría de almacenamiento.

especificadores-declaración:
especificador-categoría-almacenamiento especificadores-de-declaración$_{opt}$
especificador-de-tipo especificadores-de-declaración$_{opt}$
cualificador-de-tipo especificadores-de-declaración$_{opt}$

lista-declaradores-init:
declarador-init
lista-declaradores-init , declarador-init

declarador-init:
declarador
declarador = *inicializador*

Los declaradores se discutirán posteriormente (§A8.5), y contienen los nombres que están siendo declarados. Una declaración debe tener al menos un declarador o su especificador de tipo debe declarar el rótulo de una estructura, un rótulo de unión o los miembros de una enumeración; no se permiten declaraciones vacías.

A8.1 Especificadores de categoría de almacenamiento

Los especificadores de categoría de almacenamiento son:

especificador-categoría-almacenamiento:

```
auto
register
static
extern
typedef
```

Los significados de las categorías de almacenamiento se discutieron en §A4.

Los especificadores **auto** y **register** dan a los objetos declarados categoría de almacenamiento automático y sólo se pueden usar dentro de funciones. Tales declaraciones también sirven como definiciones y provocan que se reserve almacenamiento. Una declaración **register** es equivalente a una declaración auto, pero sugiere que se hará acceso frecuente a los objetos declarados. Pocos objetos son realmente localizados en registros y sólo ciertos tipos son elegibles; las restricciones son dependientes de la implantación. Sin embargo, si un objeto es declarado **register**, el operador unario **&** no se le puede aplicar, explícita o implícitamente.

> La regla de que es ilegal calcular la dirección de un objeto declarado **register**, pero que realmente se tomará como **auto**, es nueva.

El especificador **static** da a los objetos declarados categoría de almacenamiento estática, y se puede emplear dentro o fuera de las funciones. Dentro de una función, este especificador provoca que se asigne almacenamiento y sirve como definición; para sus efectos fuera de una función, véase §A11.2.

Una declaración con **extern**, utilizada dentro de una función, especifica que el almacenamiento para los objetos declarados está definido en algún otro lugar; para sus efectos fuera de una función, véase §A11.2.

El especificador **typedef** no reserva almacenamiento y se llama especificador de categoría de almacenamiento sólo por conveniencia sintáctica; se discute en §A8.9.

Cuando más se puede dar un especificador de categoría de almacenamiento dentro de una declaración. Si no se da ninguno, se utilizan estas reglas: los objetos declarados dentro de una función se toman como **auto**; las funciones declaradas dentro de una función se toman como **extern**; los objetos y funciones declarados fuera de una función se toman como estáticos, con liga externa. Véase §§A10-A11.

A.8.2 Especificadores de tipo

Los especificadores de tipo son

especificador-de-tipo:

```
void
char
short
int
long
float
double
signed
unsigned
```

especificador-estructura-o-unión
especificador-enum
nombre-typedef

Cuando más se puede especificar una de las palabras **long** o **short** junto con **int**; el significado es el mismo si no se menciona **int**. La palabra **long** se puede especificar junto con **double**. Cuando más se puede especificar una de las palabras **signed** o **unsigned** junto con **int** o cualquiera de sus variantes, **short**, **long** o con **char**. Cualquiera de las dos puede aparecer sola; en tal caso se entiende como **int**. El especificador **signed** es útil para forzar a los objetos **char** a tener signo; es permisible pero redundante dentro de otros tipos enteros.

De otra manera, cuando más se puede dar un especificador de tipo en una declaración. Si el especificador de tipo se omite de una declaración, se toma como **int**.

Los tipos también se pueden calificar, para indicar propiedades especiales de los objetos que están siendo declarados.

calificador-de-tipo
  const
  volatile

Los calificadores de tipo pueden aparecer con cualquier especificador de tipo. Un objeto **const** se puede inicializar, pero después no se le puede asignar nada. No hay semántica independiente de la implantación para objetos **volatile**.

> Las propiedades **const** y **volatile** son nuevas dentro del estándar ANSI. El propósito de **const** es anunciar objetos que pueden ser localizados en memoria de sólo lectura y tal vez incrementar las oportunidades para optimización. El propósito de **volatile** es forzar a la implantación a suprimir la optimización que, de otra manera, podría ocurrir. Por ejemplo, para una máquina con entrada/salida asignado a memoria, el apuntador a un registro de dispositivo se podría declarar como un apuntador a **volatile** para prevenir que el compilador remueva las referencias aparentemente redundantes a través del apuntador. Excepto que debe diagnosticar explícitamente los intentos de cambiar objetos **const**, el compilador puede ignorar estos calificadores.
> Un tercer calificador, **noalias**, permanece bajo consideración por el comité de estandarización.

A8.3 Declaraciones de estructura y unión

Una estructura es un objeto que consta de una secuencia de miembros nombrados de varios tipos. Una unión en un objeto que contiene, en momentos distintos, cualquiera de algunos miembros de varios tipos. Los especificadores de estructura y unión tienen la misma forma.

especificador-estructura-o-unión:
  estructura-o-unión identificador$_{opt}$ { *lista-declaraciones-struct* }
  estructura-o-unión identificador

estructura-o-unión:
  struct
  union

Una lista-de-declaraciones-struct es una secuencia de declaraciones para los miembros de la estructura o unión:

lista-declaraciones-struct:
  declaración-struct
  lista-declaraciones-struct declaración-struct

declaración-struct:
  lista-calificador-especificador lista-de-declaradores-struct ;

lista-cualificador-especificador:
  especificador-de-tipo lista-calificador-especificador$_{opt}$
  calificador-de-tipo lista-calificador-especificador$_{opt}$

lista-declaradores-struct:
  declarador-struct
  lista-declaradores-struct , *declarador-struct*

Por lo general, un declarador-struct es sólo un declarador para un miembro de estructura o unión. Un miembro de estructura también puede constar de un número especificado de bits. Tal miembro también se llama *campo de bits*, o simplemente *campo*; un carácter dos puntos marca el inicio de su longitud después del nombre del campo.

declarador-struct:
  declarador
  declarador$_{opt}$: *expresión-constante*

Un especificador de tipo de la forma

  estructura-o-unión identificador { *lista-declaraciones-struct* }

declara que el identificador será el *rótulo* de la estructura o unión especificado por la lista. Una declaración posterior en el mismo alcance o más interno se puede referir al mismo tipo utilizando el rótulo dentro de un especificador sin la lista:

  estructura-o-unión identificador

Si aparece un especificador con un rótulo pero sin lista cuando el rótulo no está declarado, se especifica un *tipo incompleto*. Los objetos con tipo incompleto de estructura o unión se pueden mencionar en contextos donde no sea necesario su tamaño, por ejemplo, en declaraciones (no definiciones), para estipular un apuntador o para crear un **typedef**, pero no de otra manera. El tipo se completa al presentarse un especificador subsecuente con ese rótulo que contenga una lista de declaraciones. Incluso en especificadores con una lista, el tipo de la estructura o unión que está siendo declarado es incompleto dentro de la lista, y se completa sólo en el que termina el especificador.

Una estructura no puede contener un miembro de tipo incompleto. Por lo tanto, es imposible declarar una estructura o unión que contenga una instancia de ella misma. Sin embargo, además de dar un nombre al tipo de estructura o unión, los rótulos permiten la definición de estructuras autorreferenciadas; una estructura o unión puede contener un apuntador a una instancia de ella misma, debido a que pueden ser declarados apuntadores a tipos incompletos.

Una regla muy especial se aplica a declaraciones de la forma

  estructura-o-unión identificador ;

que declara una estructura o unión, pero no tiene lista de declaraciones ni declaradores. Aun si el identificador es un rótulo de estructura o unión ya declarado en un alcance más externo (§A11.1), esta declaración hace al identificador el rótulo de una nueva estructura o unión, de tipo incompleto, en el alcance actual.

> Esta regla es nueva bajo ANSI. Su función es tratar con estructuras mutuamente recursivas declaradas en un alcance más interno, pero cuyos rótulos podrían haber sido ya declarados en el alcance más externo.

Un especificador de estructura o unión con una lista sin rótulo crea un tipo único; se le puede hacer referencia directamente sólo en la declaración de la que es parte.

Los nombres de miembros y rótulos no entran en conflicto entre ellos o con variables ordinarias. Un nombre de miembro no puede aparecer dos veces en la misma estructu-

ra o unión, pero el mismo nombre de miembro se puede emplear en diferentes estructuras o uniones.

En la primera edición de este libro, los nombres de miembros de estructuras y uniones no estaban asociados con su padre. Sin embargo, esta asociación se hizo común en compiladores mucho antes del estándar ANSI.

Un miembro que no sea campo de una estructura o unión puede tener cualquier tipo de objeto. Un miembro campo (que no requiere tener un declarador y, por tanto, puede no tener nombre) tiene tipo **int**, **unsigned int**, o **signed int**, y es interpretado como un objeto de tipo entero de la longitud en bits especificada; el que un campo **int** se trate como con signo depende de la implantación. Los campos adyacentes que son miembros de estructuras se empaquetan en unidades de almacenamiento dependientes de la implantación en una dirección también dependiente. Cuando hay la posibilidad de que un campo que sigue a otro no entre en una unidad de almacenamiento parcialmente llena, se puede separar en unidades, o la unidad se puede rellenar. Un campo sin nombre con amplitud 0 fuerza a este rellenado, de modo que el siguiente campo principiará en la orilla de la siguiente unidad de asignación.

El estándar ANSI hace que los campos sean aún más dependientes de la implantación que la primera edición. Es recomendable leer las reglas del lenguaje para almacenar campos de bits como "dependientes de la implantación" sin limitaciones. Las estructuras con campos de bits se pueden emplear como una forma transportable de intentar reducir el almacenamiento requerido para una estructura (con el costo probable de incrementar el espacio de instrucciones y tiempo para tener acceso a los campos), o como una forma no transportable de describir una plantilla de almacenamiento conocida al nivel de bits. En el segundo caso, es necesario entender las reglas de la implantación local.

Los miembros de una estructura tienen direcciones ascendentes en el orden de sus declaraciones. De una estructura un miembro que no sea campo se alinea con un límite de direccionamiento dependiendo de su tipo; por tanto, puede haber huecos sin nombre dentro de una estructura. Si un apuntador a una estructura es convertido al tipo de un apuntador a su primer miembro, el resultado se refiere al primer miembro.

Se puede pensar en una unión como una estructura donde todos sus miembros principian en el desplazamiento 0 y cuyo tamaño es suficiente para contener a cualquiera de sus miembros. Cuando más, uno de los miembros puede ser almacenado dentro de una unión a la vez. Si un apuntador a unión es convertido al tipo de un apuntador a un miembro, el resultado se refiere a ese miembro.

Un ejemplo simple de declaración de estructura es

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

que contiene un arreglo de 20 caracteres, un entero y dos apuntadores a estructuras semejantes. Una vez que se ha dado esta declaración, la declaración

```
struct tnode s, *sp;
```

declara s como una estructura de la variedad dada y **sp** como apuntador a una estructura de ese tipo. Con estas declaraciones, la expresión


```
sp->count
```

se refiere al campo **count** de la estructura a la que apunta **sp**;

```
s.left
```

se refiere al apuntador al subárbol izquierdo de la estructura **s**; y

```
s.right->tword[0]
```

se refiere al primer carácter del miembro **tword** del subárbol derecho de **s**.

En general, un miembro de una unión no puede ser inspeccionado a menos de que el valor de la unión haya sido asignado utilizando ese miembro. Sin embargo, una consideración especial simplifica el uso de uniones: si una unión contiene varias estructuras que comparten una secuencia inicial común, y si la unión actualmente contiene una de esas estructuras, se permite hacer referencia a la parte inicial común de cualesquiera de las estructuras contenidas. Por ejemplo, el siguiente fragmento es legítimo:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

A8.4 Enumeraciones

Las enumeraciones son tipos con valores que fluctúan entre un conjunto de constantes nombradas que se llaman enumeradores. La forma de un especificador de enumeración se toma de la de las estructuras y uniones.

especificador-enum:
  enum *identificador*$_{opt}$ { *lista-de-enumerador* }
  enum *identificador*
lista-de-enumerador:
  enumerador
  lista-de-enumerador **,** *enumerador*
enumerador:
  identificador
  identificador = *expresión-constante*

Los identificadores dentro de una lista de enumerador se declaran como constantes de tipo int y pueden aparecer en cualquier lugar donde se requiera una constante. Si no aparecen enumeradores con =, entonces los valores de las correspondientes constantes principian en 0 y se incrementan en 1 al leer la declaración de izquierda a derecha. Un enumerador con = da al identificador asociado el valor especificado; los identificadores subsecuentes continúan la progresión del valor asignado.

Los nombres de enumeradores en el mismo alcance deben ser distintos entre sí y de los nombres de variables ordinarias, pero no es necesario que los valores sean distintos.

El papel del identificador en el especificador-enum es análogo al del rótulo de las estructuras en un especificador-de-estructura; nombra una enumeración particular. Las reglas para especificadores-enum con y sin rótulos y listas son las mismas que para especificadores de estructuras y uniones, excepto que los tipos de enumeración incompleta no existen; el rótulo de un especificador-enum sin una lista de enumeradores debe referirse a un especificador en el alcance dentro de la lista.

Las enumeraciones son nuevas desde la primera edición de este libro, pero han sido parte del lenguaje por algunos años.

A8.5 Declaradores

Los declaradores tienen la sintaxis:

declarador
  apuntador$_{opt}$ *declarador-directo*
declarador-directo:
  identificador
  (*declarador*)
  declarador-directo [*expresión-constante*$_{opt}$]
  declarador-directo (*lista-tipos-de-parámetro*)
  declarador-directo (*lista-de-identificadores*$_{opt}$)
apuntador:
  * *lista-calificadores-de-tipo*$_{opt}$
  * *lista-calificadores-de-tipo*$_{opt}$ *apuntador*
lista-calificadores-de-tipo:
  calificador-de-tipo
  lista-calificadores-de-tipo calificador-de-tipo

La estructura de los declaradores es semejante a la de las expresiones de indirección, funciones y arreglos; el agrupamiento es el mismo.

A8.6 Significado de los declaradores

La lista de declaradores aparece después de una secuencia de especificadores de categoría de tipo y almacenamiento. Cada declarador declara un identificador principal único, que aparece como la primera alternativa de la producción para *declarador-directo*. Los especificadores de categoría de almacenamiento se aplican directamente a este identificador, pero su tipo depende de la forma de su declarador. Un declarador es leído como la afirmación de que cuando su identificador aparece en una expresión de la misma forma que el declarador, produce un objeto del tipo especificado.

Considerando sólo las partes de tipo de los especificadores de declaración (§A8.2) y un declarador particular, una declaración tiene la forma "T D", donde T es un tipo y D es un declarador. El tipo atribuido al identificador en las varias formas del declarador se describe inductivamente empleando esta notación.

En una declaración T D donde D es un identificador solo, el tipo del identificador es T.

En una declaración T D donde D tiene la forma

(D1)

el tipo del identificador en D1 es el mismo que el de D. Los paréntesis no alteran el tipo, pero pueden cambiar la asociación de declaradores complejos.

A8.6.1 Declaradores de apuntadores

Es una declaración T D en donde D tiene la forma

* *lista-calificadores-de-tipo*$_{opt}$ D1

y el tipo del identificador que está en la declaración T D1 es "*modificador-de-tipo* T", el tipo del identificador de D es "*modificador-de-tipo lista-calificadores-de-tipo* apuntador a T". Los calificadores que siguen al * se aplican al apuntador en sí, no al objeto al que apunta.

Por ejemplo, considere la declaración,

```
int *ap[];
```

Aquí ap[ ] juega el papel de D1; una declaración "int ap[ ]" (más abajo) dará a ap el tipo "arreglo de int", la lista calificador-de-tipo está vacía, y el modificador-de-tipo es "arreglo de". Por consiguiente la declaración da a ap el tipo "arreglo de apuntadores a int".

Como otros ejemplos, las declaraciones

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

declaran un entero i y un apuntador a un entero pi. El valor del apuntador constante cpi no puede ser cambiado; siempre apunta a la misma localidad, aunque el valor al que se refiere puede ser alterado. El entero ci es constante, y no se puede cambiar (aunque sí se puede inicializar, como aquí). El tipo de pci es "apuntador a const int", y pci en sí puede ser modificada para apuntar a otro lugar, pero el valor al que apunta no se puede alterar por asignación a través de pci.

A8.6.2 Declaradores de arreglos

En una declaración T D donde D tiene la forma

D1 [*expresión-constante*$_{opt}$]

y el tipo del identificador en la declaración T D1 es "*modificador-de-tipo* T", el tipo del identificador D es "*modificador-de-tipo* arreglo de T". Si la expresión-constante está presente, debe ser de tipo entero, y con valor mayor que 0. Si se omite la expresión constante que especifica el límite, el arreglo tiene tipo incompleto.

Un arreglo se puede construir a partir de un tipo aritmético, de un apuntador, de una estructura o unión o de otro arreglo (para generar un arreglo de varias dimensiones). Cualquier tipo del que se construya un arreglo debe ser completo; no debe ser un arreglo o estructura de tipo incompleto. Esto implica que para un arreglo multidimensional, sólo se puede omitir la primera dimensión. El tipo de un objeto de tipo arreglo incompleto se com-

pleta con otra declaración para el objeto (§A10.2), completa o por su inicialización (§A8.7). Por ejemplo,

```
float fa[17], *afp[17];
```

declara un arreglo de número **float** y un arreglo de apuntadores a número **float**. Por otro lado,

```
static int x3d[3][5][7];
```

declara un arreglo tridimensional estático de enteros, con rango **3 × 5 × 7**. Con todo detalle, **x3d** es un arreglo de tres elementos; cada elemento es un arreglo de cinco arreglos; cada uno de los últimos arreglos es un arreglo de siete enteros. Cualquiera de las expresiones **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** pueden aparecer razonablemente dentro de una expresión. Los primeros tres tienen tipo "arreglo" y el último tiene tipo **int**. Más específicamente, **x3d[i][j]** es un arreglo de 7 enteros, y **x3d[i]** es un arreglo de 5 arreglos de 7 enteros.

La operación de indexado de un arreglo está definida de modo que **E1[E2]** es idéntica a ***(E1+E2)**. Por lo tanto, fuera de su apariencia asimétrica, la indexación es una operación conmutativa. Debido a las reglas de conversión que se aplican a **+** y a los arreglos (§§A6.6, A7.1, A7.7), si **E1** es un arreglo y **E2** un entero, entonces **E1[E2]** se refiere al **E2**-ésimo miembro de **E1**.

En el ejemplo, **x3d[i][j][k]** es equivalente a ***(x3d[i][j] + k)**. La primera subexpresión **x3d[i][j]** se convierte por §A7.1 al tipo "apuntador a arreglo de enteros"; por §A7.7, la adición involucra multiplicación por el tamaño de un entero. De las reglas se sigue que los arreglos se almacenan por renglones (el último subíndice varía más rápido), y que el primer subíndice dentro de la declaración ayuda a determinar la cantidad de almacenamiento consumido por un arreglo, pero no tiene más utilidad en el cálculo de subíndices.

A8.6.3 Declaración de funciones

En una declaración del nuevo estilo **T D**, donde **D** tiene la forma

D1(*lista-tipos-de-parámetro*)

y el tipo del identificador dentro de la declaración **T D1** es "*modificador-de-tipo* **T**", el tipo del identificador de **D** es "*modificador-de-tipo* función con argumento *lista-tipos-de-parámetros* que regresa **T**".

La sintaxis de los parámetros es

lista-tipos-de-parámetro:
  lista-de-parámetros
  lista-de-parámetros , . . .

lista-de-parámetros:
  declaración-parámetro
  lista-de-parámetros , *declaración-parámetro*

declaración-parámetro:
  especificadores-de-declaración declarador
  especificadores-de-declaración declarador-abstracto$_{opt}$

En la declaración del nuevo estilo, la lista de parámetros estipula los tipos de los parámetros. Como un caso especial, el declarador para una función del nuevo estilo sin parámetros tiene una lista de tipos de parámetros consistente únicamente en la palabra reservada **void**. Si la lista de tipos de parámetros finaliza con puntos suspensivos ", . . .", entonces la función puede aceptar más argumentos que el número de parámetros descritos explícitamente; ver §A7.3.2.

Los tipos de parámetros que son arreglos o funciones se alteran y quedan como apuntadores, de acuerdo con las reglas para conversiones de parámetros; ver §A10.1. El único especificador de categoría de almacenamiento permitido dentro de un especificador de declaración de parámetros es **register**, y este especificador es ignorado a menos que el declarador de función encabece una definición de función. De modo semejante, si los declaradores que están en las declaraciones de parámetros contienen identificadores, y el declarador de función no encabeza una definición de función, los identificadores salen inmediatamente del alcance. Los declaradores abstractos, que no mencionan a los identificadores, se discuten en §A8.8.

En una declaración de función del estilo anterior **T D**, donde **D** tiene la forma

D1(*lista-de-identificadores$_{opt}$*)

y el tipo del identificador dentro de la declaración **T D1** es "*modificador-de-tipo* **T**", el tipo del identificador de **D** es "*modificador-de-tipo* función de argumentos no especificados que regresa **T**". Los parámetros (si están presentes) tienen la forma

lista-de-identificadores:
  identificador
  lista-de-identificadores , *identificador*

En el declarador del estilo anterior, la lista de identificadores debe estar ausente a menos de que el declarador se utilice en el encabezador de una definición de función (§A10.1). La declaración no proporciona ninguna información acerca de los tipos de los parámetros.

Por ejemplo, la declaración

```
int f(), *fpi(), (*pfi)();
```

declara una función **f** que regresa un entero, una función **fpi** que regresa un apuntador a un entero, y un apuntador **pfi** a una función que regresa un entero. En ninguna de éstas se especifica la lista de parámetros; están en estilo anterior.

En una declaración del nuevo estilo

```
int strcpy(char *dest, const char *source), rand(void);
```

strcpy es una función que regresa un **int**, con dos argumentos, el primero es un apuntador a carácter y el segundo un apuntador a caracteres constantes. Los nombres de los parámetros son realmente comentarios. La segunda función **rand** no emplea argumentos y regresa **int**.

Los declaradores de funciones con prototipos de parámetros son, con mucho, el cambio más importante introducido al lenguaje por el estándar ANSI. Ofrecen una ventaja sobre los declaradores "del estilo anterior" de la primera edición, proporcionando detección de errores y conversión de argumentos entre llamadas a funciones, pero a un costo: desorden y confusión durante su introducción, y la necesidad de permitir ambas formas. Se requirieron algunas aberraciones sintácticas para compatibilidad, como **void**, usado como una marca explícita de las funciones sin parámetros del nuevo estilo.

La notación de puntos suspensivos ",..." para funciones con número variable de argumentos también es nueva y, junto con las macros en el *header* estándar **<stdarg.h>**, formaliza un mecanismo que estuvo oficialmente prohibido pero extraoficialmente permitido en la primera edición.
Esas notaciones fueron adaptadas del lenguaje **C++**.

A8.7 Inicialización

Cuando se declara un objeto, su declarador-init puede especificar un valor inicial para el identificador que está siendo declarado. El inicializador es precedido por **=**, y es una expresión o una lista de inicializadores anidados entre llaves. Una lista puede terminar con coma, un buen detalle para un formato claro.

inicializador:
    expresión-asignación
    { *lista-de-inicializadores* }
    { *lista-de-inicializadores* , }

lista-de-inicializadores:
    inicializador
    lista-de-inicializadores , *inicializador*

Todas las expresiones del inicializador para un objeto o arreglo estático deben ser expresiones constantes tal como se describe en §A7.19. Las expresiones en el inicializador para un objeto o arreglo **auto** o **register** deben igualmente ser expresiones constantes si el inicializador es una lista encerrada entre llaves. Sin embargo, si el inicializador para un objeto automático es una expresión simple, no requiere ser una expresión constante, sino que debe tener simplemente el tipo apropiado para la asignación al objeto.

La primera edición no aprobaba la inicialización de estructuras automáticas, uniones o arreglos. El estándar ANSI lo permite, pero sólo por construcciones constantes o menos de que el inicializador se pueda expresar con una expresión simple.

Un objeto estático no inicializado explícitamente se inicializa como si a él (o a sus miembros) se le asigna la constante 0. El valor inicial de un objeto automático que no esté explícitamente inicializado es indefinido.

El inicializador para un estructura es una expresión del mismo tipo o una lista entre llaves de inicializadores para sus miembros en orden. Si hay menos inicializadores que
El inicializador para una estructura es o una expresión del mismo tipo, o una lista entre llaves de inicializadores para sus miembros en orden. Si hay menos inicializadores que los miembros de la estructura, los últimos miembros son inicializados con 0. No puede haber más inicializadores que miembros.

El inicializador para un arreglo es una lista de inicializadores entre llaves para sus miembros. **Si el arreglo tiene tamaño desconocido, el número de inicializadores determina el tamaño del arreglo y su tipo se completa. Si el arreglo tiene tamaño fijo, el número de** inicializadores no puede exceder al número de miembros del arreglo; si hay menos, los últimos miembros son inicializados con 0.

Como caso especial, un arreglo de caracteres se puede inicializar con una cadena literal; los caracteres sucesivos de la cadena inicializan miembros sucesivos del arreglo. De modo semejante, un carácter literal amplio (§A2.6) puede inicializar un arreglo de tipo **wchar_t**. **Si el arreglo tiene tamaño desconocido, el número de caracteres en la cadena, incluyendo el carácter nulo de terminación, determina su tamaño; si su tamaño es fijo, el número de caracteres de la cadena, sin contar el carácter nulo de terminación, no debe exceder al tamaño del arreglo.**

El inicializador para una unión es una expresión simple del mismo tipo o un inicializador **entre llaves para el primer miembro de la unión.**

La primera edición no permitía la inicialización para uniones. La regla del "primer miembro" es confusa, pero es difícil generalizar sin la nueva sintaxis. Además de permitir que las uniones sean inicializadas explícitamente, al menos en una forma primitiva, esta regla ANSI vuelve definitiva la semántica de uniones estáticas no inicializadas explícitamente.

Un *agregado* **es una estructura o un arreglo. Si un agregado contiene miembros de tipo** agregado, las reglas de inicialización se aplican en forma recursiva. Las llaves pueden desaparecer de la inicialización como sigue: si el inicializador para un miembro de agregado que en sí es un agregado principia con una llave izquierda, entonces la lista de inicializadores separados por coma que sigue inicializa los miembros del subagregado; es erróneo que haya más inicializadores que miembros. Sin embargo, si el inicializador para un subagregado no principia con una llave izquierda, entonces sólo se toman de la lista los elementos suficientes para los miembros del subagregado; cualesquiera miembros restantes se dejan para inicializar el siguiente miembro del agregado del cual el subagregado es parte.

Por ejemplo,

```
int x[] = { 1, 3, 5 };
```

declara e inicializa x como un arreglo de una dimensión con tres miembros, puesto que no se ha especificado tamaño y existen tres inicializadores.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

es una inicialización completamente entre llaves: 1, 3, y 5 inicializan al primer renglón del arreglo **y[0]**, es decir **y[0][0]**, **y[0][1]**, y **y[0][2]**. En igual forma, las siguientes dos líneas inicializan **y[1]** y **y[2]**. El inicializador termina antes y, por lo tanto, los elementos de **y[3]** son inicializados con 0. Precisamente el mismo efecto se puede obtener con

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

El inicializador para y principia con una llave izquierda, pero el de y[0] no, por lo que **se utilizan tres elementos de la lista. De la misma forma los siguientes tres elementos de la lista se toman sucesivamente para** y[1] **y después** y[2]. **También,**

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

inicializa la primera columna de y (considerado como un arreglo bidimensional) y deja al resto en 0.

Finalmente,

```
char msg[] = "Error de sintaxis en línea %s\n";
```

muestra un arreglo de caracteres cuyos miembros son inicializados con una cadena; su tamaño incluye el carácter nulo de terminación.

A8.8 Nombres de tipos

Dentro de muchos contextos (para especificar las conversiones de tipo explícitamente con un *cast*, para declarar tipos de parámetros en declaradores de función, y como un argumento de **sizeof**) es necesario proporcionar el nombre de un tipo de dato. Esto se logra utilizando un *nombre de tipo*, que sintácticamente es una declaración para un objeto de ese tipo, omitiendo el nombre del objeto.

nombre-de-tipo:
  lista-calificador-especificador declarador-abstracto$_{opt}$

declarador-abstracto:
  apuntador
  apuntador$_{opt}$ declarador-abstracto-directo

declarador-abstracto-directo:
  (declarador-abstracto)
  declarador-abstracto-directo$_{opt}$ [expresión-constante$_{opt}$]
  declarador-abstracto-directo$_{opt}$ (lista-tipos-de-parámetro$_{opt}$)

Es posible identificar unívocamente el lugar dentro del declarador abstracto en donde podría aparecer el identificador si la construcción fuera un declarador en una declaración. El tipo nombrado es entonces el mismo que el tipo del identificador hipotético. Por ejemplo,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

nombra respectivamente los tipos "entero", "apuntador a entero", "arreglo de 3 apuntadores a enteros", "apuntador a un arreglo de un número no especificado de enteros", "función de parámetros no especificados que regresa un apuntador a entero" y "arreglo, de tamaño no especificado, de apuntadores a funciones sin parámetros que regresa cada una un entero".

A8.9 Typedef

Las declaraciones cuyo especificador de categoría de almacenamiento es **typedef** no declaran objetos; en lugar de ello definen identificadores que nombran tipos. Esos identificadores se llaman nombres typedef.

nombre-typedef:
  identificador

Una declaración **typedef** atribuye un tipo a cada nombre entre sus declaradores en la forma usual (ver §8.6). Luego de eso, cada nombre **typedef** es sintácticamente equivalente a una palabra reservada para especificador de tipo para el tipo asociado.

Por ejemplo, después de

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

las construcciones

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

son declaraciones legítimas. El tipo de **b** es **long**, el de **bp** es "apuntador a **long**", y el de **z** es la estructura especificada; **zp** es un apuntador a tal estructura.

typedef no introduce nuevos tipos, sólo formas sinónimas para tipos que se podrían mencionar en otra forma. En el ejemplo, **b** tiene el mismo tipo que cualquier otro objeto **long**.

Los nombres **typedef** se pueden redeclarar dentro de un alcance más interno, pero se debe dar un conjunto no vacío de especificadores de tipo. Por ejemplo,

```
extern Blockno;
```

no redeclara a **Blockno**, pero

```
extern int Blockno;
```

sí lo hace.

A8.10 Equivalencia de tipo

Dos listas de especificadores de tipo son equivalentes si contienen el mismo conjunto de especificadores de tipo, tomando en cuenta que algunos especificadores pueden implicar otros (por ejemplo, **long** solo implica **long int**). Estructuras, uniones, y enumeraciones con rótulos diferentes son distintas, y una unión, estructura, o enumeración sin rótulo estipula un tipo único.

Dos tipos son el mismo si sus declaradores abstractos (§A8.8), después de expandir cualesquiera tipos **typedef** y de eliminar cualesquiera identificadores de parámetros de función, son las mismas o equivalentes listas de especificadores de tipo. Los tamaños de los arreglos y los parámetros de las funciones son significativos.

A9. Proposiciones

Excepto en donde así se describe, las proposiciones se ejecutan en secuencia. Las proposiciones se ejecutan por sus efectos y no tienen valores. Entran en varios grupos.

proposición:
  proposición-etiquetada
  proposición-expresión
  proposición-compuesta
  proposición-de-selección
  proposición-de-iteración
  proposición-de-salto

A9.1 Proposiciones etiquetadas

Las proposiciones pueden tener etiquetas como prefijos.

proposición-etiquetada:
  identificador : proposición
  case *expresión-constante : proposición*
  default *: proposición*

Una etiqueta consistente en un identificador declara al identificador. El único uso de una etiqueta identificadora es como destino de un goto. El alcance del identificador está dentro de la función actual. Debido a que las etiquetas tienen su propio espacio de nombre, no interfieren con otros identificadores y no se pueden redeclarar. Véase §A11.1.

Las etiquetas de **case** y **default** se aplican con la proposición **switch** (§A9.4). La expresión constante del **case** debe tener tipo entero.

Las etiquetas por sí mismas no alteran el flujo de control.

A9.2 Proposición de expresión

La mayoría de las proposiciones son proposiciones de expresión, que tienen la forma

proposición-expresión:
  $expresión_{opt}$;

La mayoría de las proposiciones expresiones son asignaciones o llamadas a función. Todos los efectos colaterales de la expresión son completados antes de ejecutar la siguiente proposición. Si la expresión se omite, la construcción se llama proposición nula; a menudo se emplea para proporcionar un cuerpo vacío a una proposición de iteración o para situar una etiqueta.

A9.3 Proposición compuesta

Para que se puedan utilizar varias proposiciones donde se espera una, se proporciona la proposición compuesta (también llamada "bloque"). El cuerpo de una definición de función es una proposición compuesta.

proposición-compuesta:
  { $lista\text{-}declaración_{opt}$ $lista\text{-}de\text{-}proposiciones_{opt}$ }

lista-de-declaraciones:
  declaración
  lista-de-declaraciones declaración

lista-de-proposiciones:
  proposición
  lista-de-proposiciones proposición

Si un identificador dentro de la lista-de-declaraciones estuvo en un alcance exterior al bloque, la declaración más externa se suspende dentro del bloque (véase §A11.1), después de lo cual continúa. Un identificador se puede declarar sólo una vez dentro del mismo bloque. Estas reglas se aplican a identificadores dentro del mismo espacio de nombre (§A11); los identificadores dentro de espacios de nombre diferentes son tratados como distintos.

La inicialización de objetos automáticos se realiza cada vez que se entra al bloque por la parte superior, y procede en el orden de los declaradores. Si se ejecuta un salto dentro del bloque, estas inicializaciones no se realizan. Las inicializaciones de objetos static se realizan sólo una vez, antes de que el programa comience su ejecución.

A9.4 Proposiciones de selección

Las proposiciones de selección eligen uno de varios flujos de control.

proposición-de-selección:
  if (*expresión*) *proposición*
  if (*expresión*) *proposición* else *proposición*
  switch (*expresión*) *proposición*

En ambas formas de la proposición **if**, la expresión, que debe ser aritmética o de tipo apuntador, es evaluada, incluyendo todos los efectos colaterales, y si se compara como diferente de 0, se ejecuta la primera subproposición. En la segunda forma, la segunda subproposición se ejecuta si la expresión es 0. La ambigüedad del **else** se resuelve conectando un else con el último if sin else encontrado en el mismo nivel de anidamiento del bloque.

La proposición **switch** provoca que el control sea transferido a una de varias proposiciones, dependiendo del valor de una expresión, que debe tener tipo entero. La subproposición controlada por un **switch** usualmente es compuesta. Cualquier proposición dentro de la subproposición se puede etiquetar con una o más etiquetas **case** (§A9.1). La expresión de control tiene promoción integral (§A6.1), y las constantes de los **case** son convertidas al tipo promovido. No se permite que dos de las expresiones **case** asociadas con el mismo **switch** puedan tener el mismo valor después de la conversión. Cuando más puede haber una etiqueta **default** asociada con un **switch**. Las proposiciones **switch** pueden estar anidadas; una etiqueta **case** o **default** está asociada con el **switch** más anidado que la contiene.

Cuando se ejecuta la proposición **switch**, su expresión se evalúa incluyendo todos los efectos colaterales y es comparada con cada constante del **case**. Si una de las constantes **case** es igual al valor de la expresión, el control pasa a la proposición de la etiqueta **case** coincidente. Si ninguna constante **case** coincide con la expresión y si existe una etiqueta **default**, el control pasa a la proposición etiquetada. Si ningún caso coincide y no hay **default**, entonces no se ejectua ninguna de las subproposiciones del **switch**.

En la primera edición de este libro, se requería que la expresión de control del **switch** y las constantes **case** tuvieran tipo **int**.

A9.5 Proposiciones de iteración

Las proposiciones de iteración especifican la ejecución de un ciclo.

proposición-de-iteración:
  while (*expresión*) *proposición*
  do *proposición* **while** (*expresión*) ;
  for ($expresión_{opt}$; $expresión_{opt}$; $expresión_{opt}$) *proposición*

En las proposiciones **while** y **do**, la subproposición se ejecuta en forma repetida mientras que el valor de la expresión permanezca diferente de 0; la expresión debe tener tipo aritmético o apuntador. Con **while**, la prueba, incluyendo todos los efectos colaterales de

la expresión, ocurre antes de cada ejecución de la proposición; con **do**, la prueba sigue de cada iteración.

En la proposición **for**, la primera expresión se evalúa una vez y especifica la inicialización para el ciclo. No hay restricción en cuanto a su tipo. La segunda expresión debe tener tipo aritmético o apuntador; se evalúa antes de cada iteración y, si es igual a 0, el **for** termina. La tercera expresión se evalúa antes de cada iteración y especifica una reinicialización para el ciclo. No hay restricción en cuanto a su tipo. Los efectos colaterales de cada expresión se completan inmediatamente después de su evaluación. Si la subproposición no contiene **continue**, una proposición

for (*expresión1* ; *expresión2* ; *expresión3*) *proposición*

es equivalente a

```
expresión1 ;
while ( expresión2 ) {
    proposición
    expresión3 ;
}
```

Cualquiera de las tres expresiones se puede descartar. La falta de una segunda expresión hace que la prueba implicada sea equivalente a probar una constante diferente de cero.

A9.6 Proposiciones de salto

Las proposiciones de salto transfieren el control incondicionalmente.

proposición-de-salto:
```
    goto identificador ;
    continue ;
    break ;
    return expresión_opt ;
```

En la proposición **goto**, el identificador debe ser una etiqueta (§A9.1) localizada en la función actual. El control se transfiere a la proposición etiquetada.

Una proposición **continue** sólo puede aparecer dentro de una proposición de iteración, y ocasiona que el control pase a la porción de continuación del ciclo más anidado que encierra a tal proposición. En forma más precisa, dentro de cada una de las proposiciones

```
while (...) {         do {                 for (...) {
   ...                   ...                  ...
contin: ;             contin: ;            contin: ;
}                     } while (...);       }
```

un **continue** que no esté contenido dentro de una proposición de iteración más anidada es lo mismo que **goto contin**.

Una proposición **break** puede aparecer sólo dentro de una proposición de iteración o de una proposición **switch**, y termina la ejecución de lo más anidado que encierre tal proposición; el control pasa a la proposición que sigue a la proposición terminada.

Una función regresa a quien la invocó con la proposición **return**. Cuando **return** es seguido por una expresión, el valor se regresa al invocador de la función. La expresión se convierte, como si se asignara, al tipo regresado por la función en la que aparece.

Que el flujo llegue hasta el final de la función es equivalente a un **return** sin expresión. En cualquier caso, el valor regresado está indefinido.

A10. Declaraciones externas

La unidad de entrada proporcionada al compilador de C se llama unidad de traducción; consiste en una secuencia de declaraciones externas, que son declaraciones o definiciones de función.

unidad-de-traducción:
  declaración-externa
  unidad-de-traducción declaración-externa
declaración-externa:
  definición-de-función
  declaración

El alcance de las declaraciones externas persiste hasta el final de la unidad de traducción en la que son declaradas, precisamente como el efecto de las declaraciones dentro de los bloques persiste hasta el final del bloque. La sintaxis de las declaraciones externas es la misma que para todas las declaraciones, excepto que el código de las funciones sólo se puede dar en este nivel.

A10.1 Definición de funciones

Las definiciones de funciones tienen la forma

definición-de-función:
especificadores-de-declaración$_{opt}$ *declarador lista-de-declaraciones*$_{opt}$ *proposición-compuesta*

Los únicos especificadores de categoría de almacenamiento permitidos entre los especificadores de declaración son **extern** o **static**; véase §A11.2 para la distinción entre ellos.

Una función puede regresar un tipo aritmético, una estructura, una unión, un apuntador o **void**, pero no una función o un arreglo. El declarador en la declaración de una función debe especificar explícitamente que el identificador declarado tiene tipo función; esto es, debe contener una de las formas (véase §A8.6.3)

declarador-directo (*lista-tipos-de-parámetros*)
declarador-directo (*lista-de-identificadores*$_{opt}$)

donde el declarador-directo es un identificador o un identificador con paréntesis. En particular, no debe adquirir el tipo de función por medio de un **typedef**.

En la primera forma, la definición es una función en el estilo nuevo, y sus parámetros, junto con sus tipos, son declarados dentro de su lista de tipos de parámetros; la lista-de-declaraciones que sigue al declarador de la función debe estar ausente. Salvo que la lista de tipos de parámetros consista solamente en **void**, mostrando que la función no tiene parámetros, cada declarador que esté en la lista de tipos de parámetros debe contener un identificador. Si la lista de tipos de parámetros termina con ", . . ." entonces la función se puede llamar con más argumentos que parámetros; para hacer referencia a los argumentos extra se debe utilizar el mecanismo de la macro **va_arg** definido en el *header* estándar

<**stdarg.h**> y descrito en el apéndice B. Las funciones con número variable de argumentos deben tener al menos un parámetro nombrado.

En la segunda forma, la definición está en el estilo anterior: la lista de identificadores nombra los parámetros, en tanto que la lista de declaraciones les atribuye tipos. Si no se da ninguna declaración para un parámetro, su tipo se toma como **int**. La lista de declaraciones debe declarar solamente parámetros nombrados en la lista; no está permitida la inicialización y el único especificador de categoría de almacenamiento posible es **register**.

En ambos estilos de definición de funciones, se entiende que los parámetros serán declarados precisamente después del principio de la proposición compuesta que constituye el cuerpo de la función y no pueden ser redeclarados allí los mismos identificadores (aunque podrían, como otros identificadores, ser redeclarados en bloques más internos). Si se declara que un parámetro tendrá tipo "arreglo de *tipo*", la declaración se ajusta para ser "apuntador a *tipo*"; de manera semejante, si un parámetro es declarado con tipo "función que regresa *tipo*", la declaración se ajusta como "apuntador a función que regresa *tipo*". Durante la llamada a una función, los argumentos se convierten de acuerdo con las necesidades y son asignados a los parámetros; véase A7.3.2.

> La definición de funciones en el nuevo estilo es nueva en el estándar ANSI. También hay un pequeño cambio en los detalles de la promoción; la primera edición estipula que las declaraciones de parámetros **float** se ajustaron para leerse como **double**. La diferencia es notoria cuando dentro de una función se genera un apuntador a un parámetro.

Un ejemplo completo de una función en el estilo nuevo es

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Aquí **int** es el especificador de la declaración; **max(int a, int b, int c)** es el declarador de la función, y { . . . } es el bloque con el código para la función. La correspondiente definición en el estilo anterior sería

```
int max(a, b, c)
int a, b, c;
{
    /* ... */
}
```

donde ahora **int max(a, b, c)** es el declarador, e **int a, b, c;** es la lista de declaraciones para los parámetros.

A10.2 Declaraciones externas

Las declaraciones externas estipulan las características de objetos, funciones y otros identificadores. El término "externa" se refiere a su localización fuera de las funciones, y no está directamente conectado con la palabra reservada **extern**; la categoría de almacenamiento para un objeto declarado externamente puede dejarse vacía, o se puede especificar como **extern** o **static**.

Pueden existir muchas declaraciones externas para el mismo identificador dentro de la misma unidad de traducción si corresponden con su tipo y liga, y si hay cuando más una definición para el identificador.

Dos declaraciones para un objeto o función se hacen coincidir en tipo bajo las reglas discutidas en §A8.10. Además, si las declaraciones difieren debido a que un tipo es una estructura, unión o enumeración incompleta (§A8.3) y el otro es el tipo completo correspondiente con el mismo rótulo, se considera que los tipos coinciden. Más aún, si uno es un tipo de arreglo incompleto (§A8.6.2) y el otro es un tipo de arreglo completo, los tipos, si por lo demás son idénticos, también se consideran coincidentes. Finalmente, si un tipo estipula una función en el estilo anterior y el otro una función en el nuevo y por lo demás idéntica, con declaración de parámetros, los tipos se toman como coincidentes.

Si la primera declaración externa para una función u objeto incluye el especificador **static**, éste tiene *liga interna*; de otra manera tiene *liga externa*. La liga se discute en §A11.2.

Una declaración externa para un objeto es una definición si tiene inicializador. La declaración de un objeto externo que no tiene inicializador y no contiene el especificador declaración de objeto externo que no tiene un inicializador y no contiene al especificador **extern**, es una *definición tentativa*. Si en una unidad de traducción aparece una definición ..ra un objeto, cualesquiera declaraciones tentativas se tratan simplemente como declaraciones redundantes. Si no aparece ninguna definición para el objeto dentro de la unidad de traducción, todas sus definiciones tentativas se convierten en una sola definición con inicializador 0.

Cada objeto debe tener exactamente una definición. Para objetos con liga interna, esta regla se aplica en forma separada para cada unidad de traducción, debido a que los objetos ligados internamente son únicos para una unidad de traducción. Para objetos con liga externa se aplica al programa completo.

> Aunque la regla de una sola definición está formulada algo diferente en la primera edición de este libro, es en efecto igual a la que se establece aquí. Algunas implantaciones la aligeran, generalizando la idea de definición tentativa. En la formulación alterna, que es común en sistemas UNIX y reconocida como una extensión común por el estándar, todas las definiciones tentativas para un objeto ligado externamente, a través de todas las unidades de traducción de un programa, se consideran juntas en lugar de separadamente en cada unidad de traducción. Si en algún lugar del programa ocurre una definición, entonces las definiciones tentativas son simples declaraciones, pero si no aparece ninguna definición, entonces todas sus definiciones tentativas se hacen definiciones con inicializador 0.

A11. Alcance y liga

No es necesario que un programa se compile todo a la vez: el texto fuente se puede mantener en varios archivos con unidades de traducción, y se pueden cargar rutinas precompiladas de bibliotecas. La comunicación entre las funciones de un programa puede llevarse a cabo tanto a través de llamadas como a través de la manipulación de datos externos.

Por tanto, existen dos clases de alcance a considerar: primera, el *alcance léxico* de un identificador, que es la región del texto del programa dentro de la que se entienden las características del identificador; y segunda, el alcance asociado con objetos y funciones con liga externa, que determina la conexión entre identificadores en unidades de traducción compiladas por separado.

A11.1 Alcance léxico

Los identificadores se dividen en varios espacios de nombre que no interfieren entre sí; el mismo identificador se puede utilizar para diferentes propósitos, incluso dentro del mismo alcance, si los usos son en diferentes espacios de nombre. Estas categorías son: objetos, funciones, nombres typedef y constantes **enum**; etiquetas; rótulos de estructura, uniones y enumeraciones; y miembros de cada estructura o unión individualmente.

> Estas reglas difieren en varias formas de las descritas en la primera edición de este manual. Las etiquetas no tenían anteriormente su propio espacio de nombre; los rótulos de estructuras y de uniones tenían cada uno su espacio separado, y en algunas implantaciones también lo tenían los rótulos de enumeraciones; colocar las diferentes clases de rótulos en el mismo espacio es una nueva restricción. El más importante alejamiento de la primera edición es que cada estructura o unión crea un espacio de nombre separado para sus miembros, de modo que en varias estructuras puede aparecer el mismo nombre. Esta regla ha sido una práctica común por muchos años.

El alcance léxico del identificador de un objeto o función dentro de una declaración externa principia al final de su declarador y persiste hasta el final de la unidad de traducción en la que aparece. El alcance del parámetro de una definición de función principia al inicio del bloque que define la función, y persiste a través de la función; el alcance de un parametro en la declarción de la función termina al final del declarador. El alcance de un identificador declarado a la cabeza de un bloque principia al final de su de**clarador y persiste hasta el final del bloque. El alcance de una etiqueta es la totalidad de la función en la cual aparece. El alcance del rótulo de una estructura, unión o enumeración, o de una constante de enumeración, principia al aparecer en un especificador de tipo** y persiste hasta el final de la unidad de traducción (para declaraciones en el nivel externo) **o hasta el final del bloque (para declaraciones dentro de una función).**

Si un identificador se declara explícitamente a la cabeza de un bloque, incluyendo el que constituye a la función, cualquier declaración del identificador fuera del mismo bloque se suspende hasta el final.

A11.2 Liga

Dentro de una unidad de traducción, todas las declaraciones del mismo identificador de objeto o función con liga interna se refieren a la misma cosa, y el objeto o función es único para esa unidad de traducción. Todas las declaraciones para el mismo identificador de objeto o función con liga externa se refieren a la misma cosa, y el objeto o función es compartido por todo el programa.

Como se discutió en §A10.2, la primera declaración externa para un identificador da al identificador liga interna si se usa el especificador **static**, de otra manera, le da liga externa. Si la declaración para un identificador dentro de un bloque no incluye el especificador **extern**, el identificador no tiene liga y es único para la función. Si incluye **extern** y hay una declaración externa activa para el identificador dentro del alcance que rodea al bloque, entonces el identificador tiene la misma liga que la declaración externa y se refiere al mismo objeto o función; pero si no hay ninguna declaración externa visible, su liga es externa.

A12. Preprocesamiento

Un preprocesador realiza macrosubstituciones, compilación condicional e inclusión de archivos nombrados. Las líneas que principian con #, aunque estén precedidas por espacios en blanco, se comunican con este preprocesador. La sintaxis de estas líneas es independiente del resto del lenguaje; pueden aparecer en cualquier lugar y su efecto termina (independientemente del alcance) hasta el final de la unidad de traducción. Los límites de las líneas son significativos; cada línea se analiza individualmente (pero véase §A12.2 cómo unir líneas). Para el preprocesador, un *token* es un *token* del lenguaje o una secuencia de caracteres que da un nombre de archivo como en la directiva **#include** (§A12.4); además, cualquier carácter que no esté definido de otra manera se toma como *token*. Sin embargo, el efecto de los caracteres espacio en blanco que no sean espacio y tabulador horizontal está definido dentro de las líneas del preprocesador.

El preprocesamiento sucede en varias fases lógicamente sucesivas que, en una im**plantación en particular, se pueden condensar.**

1. **Primero, las secuencias trigráficas que se describen en §A12.1 son reemplazadas por** sus equivalentes. De requerirlo el medio ambiente del sistema operativo, se introducen caracteres nueva línea entre las líneas del archivo fuente.
2. Cada ocurrencia de un carácter diagonal inversa / seguido por una nueva línea se elimina, uniendo así líneas (§A12.2).
3. El programa se divide en *tokens* separados por caracteres espacio en blanco; los comen**tarios se reemplazan por un solo espacio. Después se obedecen las directivas de preprocesamiento, y las macros (§§A12.3-A12.10) se expanden.**
4. Las secuencias de escape que están dentro de caracteres y cadenas literales constantes **(§§A2.5.2, A2.6) se reemplazan por sus equivalentes; después se concatenan las cadenas literales adyacentes.**
5. **El resultado se traduce, después se liga junto con otros programas y bibliotecas, recolectando los programas y datos necesarios y conectando las funciones y objetos externos con sus definiciones.**

A12.1 Secuencias trigráficas

El conjunto de caracteres de los programas fuente de C está contenido dentro del código ASCII de siete bits, pero es un superconjunto del ISO 646-1983 *Invariant Code Set*. Para **poder representar a los programas en el conjunto reducido, todas las ocurrencias de las siguientes secuencias trigráficas se reemplazan por el carácter simple correspondiente. Este reemplazo ocurre antes de cualquier otro procesamiento.**

```
??=  #        ??(  [        ??<  {
??/  \        ??)  ]        ??>  }
??'  ^        ??!  |        ??-  ~
```

No ocurre ningún otro reemplazo.

> Las secuencias trigráficas son nuevas en el estándar ANSI.

A12.2 Unión de líneas

Las líneas que terminan con el carácter diagonal invertida / se empalman eliminando la diagonal inversa y el siguiente carácter nueva línea. Esto ocurre antes de hacer la división en *tokens*.

A12.3 Definición y expansión de macros

Una línea de control de la forma

define *identificador secuencia-de-tokens*

hace que el preprocesador reemplace las instancias subsecuentes del identificador con la secuencia de *tokens* dada; se descartan los espacios en blanco que se encuentran alrededor de la secuencia de *tokens*. Un segundo # define para el mismo identificador es erróneo a menos que la segunda secuencia de *tokens* sea idéntica a la primera, en donde todos los espacios en blanco se consideran equivalentes.

Una línea de la forma

define *identificador* (*lista-de-identificadores*) *secuencia-de-tokens*

donde no hay espacio en blanco entre el primer identificador y el (, es una macrodefinición con parámetros dados por la lista de identificadores. Tal como la primera forma, los espacios en blanco alrededor de la secuencia de *tokens* se descartan, y la macro puede redefinirse sólo con una definición en la que el número y descripción de parámetros y la secuencia de *tokens* sea idéntica.

Una línea de control de la forma

undef *identificador*

hace que el preprocesador olvide la definición del identificador. No es erróneo aplicar **#undef** a un identificador desconocido.

Cuando una macro se ha definido en la segunda forma, las instancias textuales posteriores del identificador de la macro seguidas por espacio en blanco optativo, y después por (, una secuencia de *tokens* separados por comas y un), constituyen una llamada a la macro. Los argumentos de la llamada son las secuencias de *tokens* separados por comas; las comas que se encuentran entre comillas o protegidas por paréntesis anidados no separan argumentos. Durante la recolección los argumentos no son macroexpandidos. El número de argumentos en la llamada debe coincidir con el número de parámetros en la definición. Después de que los argumentos se aíslan, se remueven los espacios en blanco que los principian y finalizan. Después se substituye la secuencia de *tokens* resultante de cada argumento por cada ocurrencia no entrecomillada del identificador del parámetro correspondiente en la secuencia de reemplazo de *tokens* de la macro. A menos de que el parámetro en la secuencia de reemplazo esté precedido por #, o precedido o seguido por ##, los *tokens* argumentos se examinan para macrollamadas y se expanden como sea necesario, justo antes de la inserción.

Dos operadores especiales influyen sobre el proceso de reemplazo. Primero, si la ocurrencia de un parámetro en la secuencia de *tokens* de reemplazo está precedida inmediatamente por #, se colocan comillas (") alrededor del parámetro correspondiente y después, tanto el # como el identificador del parámetro, se reemplazan por el argumento entrecomillado. Antes de cada carácter " o \ que aparezca alrededor o dentro de una cadena literal o constante de carácter en el argumento, se inserta un carácter /.

Segundo, si la secuencia de definición de *tokens* para cualquier clase de macro contiene un operador ##, entonces justo después del reemplazo de los parámetros, se elimina cada ##, junto con cualquier espacio en blanco de ambos lados, para concatenar los *tokens* adyacentes y formar uno nuevo. El efecto está indefinido si se producen *tokens* inválidos, o si el resultado depende del orden de procesamiento de los operadores ##. Además, ## no puede aparecer al principio o fin de una secuencia de *tokens* de reemplazo.

En ambas clases de macro, la secuencia de reemplazo de *tokens* se rastrea nuevamente en forma repetida, buscando más definiciones de identificadores. Sin embargo, una vez que un identificador dado ha sido reemplazo en una expansión dada, no se reemplaza si se encuentra nuevamente durante la búsqueda, sino que permanece sin cambio.

Aun cuando el valor final de una macroexpansión principie con #, no se toma como una directiva de preprocesamiento.

> Los detalles del proceso de la macroexpansión se describen en forma más precisa en el estándar ANSI que en la primera edición. El cambio más importante es la adición de los operadores # y ##, que hacen admisible el entrecomillado y la concatenación. Algunas de las nuevas reglas, especialmente las que involucran concatenación, son muy extrañas. (Véanse los ejemplos siguientes.)

Por ejemplo, esto se puede utilizar para "constantes manifiestas", como en

```
#define TABSIZE 100
int table[TABSIZE];
```

La definición

```
#define ABSDIFF(a, b)   ((a)>(b) ? (a)-(b) : (b)-(a))
```

define una macro que regresa el valor absoluto de la diferencia entre sus argumentos. A diferencia de la función que hace la misma actividad, los argumentos y el tipo regresado pueden tener cualquier tipo aritmético o incluso ser apuntadores. Los argumentos que pueden tener efectos colaterales son evaluados dos veces, una para la prueba y otra para producir el valor.

Dada la definición

```
#define tempfile(dir)    #dir "/%s"
```

la macrollamada **tempfile(/usr/tmp)** entrega

```
"/usr/tmp" "/%s"
```

que posteriormente se concatenará como una cadena sencilla. Después de

```
#define cat(x, y)    x ## y
```

la llamada **cat(var, 123)** produce **var123**. Sin embargo, la llamada **cat(cat(1,2), 3))** está indefinida: la presencia de ## impide que los argumentos de la llamada más externa sean expandidos. Así se produce la cadena de *tokens*.

```
cat  (  1  ,  2  )3
```

y)3 (la unión del último *token* del primer argumento con el primer *token* del segundo) no es un *token* legal. Si se introduce un segundo nivel de macrodefinición,

```
#define xcat(x,y)     cat(x,y)
```

las cosas trabajan más suavemente; **xcat(xcat(1, 2), 3)** produce 123, debido a que la expansión de **xcat** en sí no involucra al operador ##.

En la misma forma, **ABSDIFF(ABSDIFF(a,b),c)** produce lo esperado, un resultado completamente expandido.

A12.4 Inclusión de archivos

Una línea de control de la forma

include <*nombre-de-archivo*>

ocasiona el reemplazo de esa línea por el contenido completo del archivo *nombre-de-archivo*. Los caracteres del nombre *nombre-de-archivo* no deben incluir > o nueva línea y está indefinido el efecto si contiene ", ', \, o /*. El archivo nombrado se busca en una secuencia de lugares dependiendo de la implantación.

De modo semejante, una línea de control de la forma

include "*nombre-de-archivo*"

busca primero en asociación con el archivo fuente original (fase deliberadamente dependiente de la implantación), y si tal búsqueda falla, entonces lo hace como en la primera forma. El efecto de usar ', \, o /* en el nombre del archivo permanece indefinido, pero está permitido >.

Finalmente, una directiva de la forma

include *secuencia-de-tokens*

que no coincida con una de las formas previas se interpreta expandiendo la secuencia de *tokens* como texto normal; debe resultar una de las dos formas con <...> o "...", y entonces se trata como se describió anteriormente.

Los archivos #include pueden estar anidados.

A12.5 Compilación condicional

Parte de un programa se pueden compilar condicionalmente, de acuerdo con la siguiente sintaxis esquemática.

preprocesador-condicional:
    línea-if texto partes-elif parte-else$_{opt}$ #endif

línea-if:
    # if *expresión-constante*
    # ifdef *identificador*
    # ifndef *identificador*

partes-elif:
    línea-elif texto
    partes-elif$_{opt}$

línea-elif:
    # elif *expresión-constante*

parte-else:
    línea-else texto

línea-else:
    #else

Cada una de las directivas (línea-if, línea-elif, línea-else, y #endif) aparece sola en una línea. Las expresiones constantes que están en #if y posteriores líneas #elif se evalúan en orden hasta que se encuentra una expresión con valor diferente de cero; el texto que sigue a una línea con valor cero se descarta. El texto que sigue a una línea de directiva con éxito se trata normalmente. Aquí "texto" se refiere a cualquier material, incluyendo líneas del preprocesador, que no es parte de la estructura condicional; puede ser vacío. Una vez que se ha encontrado una línea #if o #elif con éxito y se ha procesado su texto, las líneas #elif y #else que le siguen, junto con su texto, se descartan. Si todas las expresiones son cero y hay un #else, el texto que sigue al #else se trata normalmente. El texto controlado por ramificaciones inactivas de la condicional se ignora, excepto para verificar el anidamiento de condicionales.

La expresión constante en #if y #elif está sujeta a macroreemplazo ordinario. Además, cualesquier expresiones de la forma

defined *identificador*

o

defined (*identificador*)

se reemplazan, antes de buscar macros, por 1L si el identificador está definido en el preprocesador y por 0L si no lo está. Cualesquiera identificadores restantes después de la macroexpansión se reemplazan por 0L. Finalmente, cada constante entera se considera como con sufijo L, de modo que toda la aritmética se forma como long o unsigned long.

La expresión constante que resulta (§A7.19) está restringida: debe ser entera y no debe contener sizeof, o una constante de enumeración.

Las líneas de control

#ifdef *identificador*
#ifndef *identificador*

son equivalentes a

if defined *identificador*
if ! defined *identificador*

respectivamente.

#elif es nueva desde la primera edición aunque ha estado disponible en algunos preprocesadores. El operador defined del preprocesador también es nuevo.

A12.6 Control de línea

Para beneficio de otros preprocesadores que generan programas en C, una línea en una de las formas

line constante "*nombre-de-archivo*"
line *constante*

ocasiona que el compilador suponga, para propósitos de diagnóstico de errores, que el número de línea de la siguiente línea fuente está dado por la constante entera decimal, y que el archivo actual de entrada está nombrado por el identificador. Si el nombre de archivo entrecomillado está ausente, el nombre recordado no cambia. Las macros de la línea son expandidas antes de ser interpretadas.

A12.7 Generación de errores

Una línea del preprocesador de la forma

**error** *secuencia-de-tokens*$_{opt}$

ocasiona que el preprocesador escriba un mensaje de diagnóstico que incluye la secuencia de *tokens*.

A12.8 Pragmas

Una línea de control de la forma

**pragma** *secuencia-de-tokens*$_{opt}$

ocasiona que el preprocesador realice una acción que depende de la implantación. Un pragma no reconocido es ignorado.

A12.9 Directiva nula

Una línea del preprocesador de la forma

#

no tiene efecto.

A12.10 Nombres predefinidos

Varios identificadores están predefinidos, y se expanden para producir información especial, ni ellos ni el operador de expresión del preprocesador **defined**, pueden estar sin definición o redefinidos.

`__LINE__`	Constante decimal que contiene el número de línea actual.
`__FILE__`	Cadena literal que contiene el nombre del archivo que se está compilando.
`__DATE__`	Cadena literal que contiene la fecha de la compilación, en la forma **"Mmm dd aaaa"**.
`__TIME__`	Cadena literal que contiene la hora de la compilación, en la forma **"hh:mm:ss"**.
`__STDC__`	La constante 1. Este identificador será definido como 1 sólo en implantaciones que conforman el estándar.

#error y **#pragma** son nuevos con el estándar ANSI; las macros predefinidas del preprocesador son nuevas, pero algunas de ellas han estado disponibles en algunas implantaciones.

A13. Gramática

A continuación se encuentra una recapitulación de la gramática expuesta a lo largo de la primera parte de este apéndice. Tiene exactamente el mismo contenido, pero se encuentra en diferente orden.

La gramática ha retirado la definición a los símbolos terminales *constante-entera, constante-de-carácter, constante-flotante, identificador, cadena,* y *constante-enumeración*; las palabras en estilo mecanográfico y los símbolos son terminales dadas literalmente. La gramática se puede transformar mecánicamente en una entrada aceptable por un generador de *parsers* automático. Además de agregar alguna marca sintáctica para indicar alternativas en las producciones, es necesario expandir las construcciones "uno de" y (dependiendo de las reglas del generador de *parsers*) duplicar cada producción con símbolo *opt*, una vez con el símbolo y otra sin él. Con un cambio adicional, borrar la producción *nombre-typedef:identificador* y hacer a *nombre-typedef* símbolo terminal, esta gramática es aceptable para el generador de *parsers* YACC. Sólo tiene un conflicto, generado por la ambigüedad del **if-else**.

unidad-de-traducción:
    declaración-externa
    unidad-de-traducción declaración-externa
declaración-externa:
    definición-de-función
    declaración
definición-de-función:
    especificadores-de-declaración$_{opt}$ *declarador* *lista-de-declaraciones*$_{opt}$
    proposición-compuesta
declaración:
    especificadores-de-declaración lista-declaradores-init$_{opt}$ **;**
lista-de-declaraciones:
    declaración
    lista-de-declaraciones declaración
especificadores-de-declaración:
    especificador-categoría-almacenamiento especificadores-de-declaración$_{opt}$
    especificador-de-tipo especificadores-de-declaración$_{opt}$
    calificador-de-tipo especificadores-de-declaración$_{opt}$
especificador-categoría-almacenamiento: uno de
    auto register static extern typedef
especificador-de-tipo: uno de
    void char short int long float double signed
    unsigned *especificador-estructura-unión*
    especificador-enum nombre-typedef
calificador-de-tipo: uno de
    const volatile
especificador-estructura-o-unión:
    estructura-o-unión identificador$_{opt}$ **{** *lista-declaraciones-struct* **}**
    estructura-o-unión identificador
estructura-o-unión: uno de
    struct union
lista-declaraciones-struct:
    declaración-struct
    lista-declaraciones-struct declaración-struct
lista-declaradores-init:
    declarador-init

```
        lista-declaradores-init , declarador-init
declarador-init:
        declarador
        declarador = inicializador
declaración-struct:
        lista-calificador-especificador lista-declaradores-struct ;
lista-calificador-especificador:
        especificador-de-tipo lista-calificador-especificador_opt
        calificador-de-tipo lista-calificador-especificador_opt
lista-declaradores-struct:
        declarador-struct
        lista-declaradores-struct , declarador-struct
declarador-struct:
        declarador
        declarador_opt : expresión-constante
especificador-enum:
        enum identificador_opt { lista-de-enumerador }
        enum identificador
lista-de-enumerador:
        enumerador
        lista-de-enumerador , enumerador
enumerador:
        identificador
        identificador = expresión-constante
declarador:
        apuntador_opt declarador-directo
declarador-directo:
        identificador
        ( declarador )
        declarador-directo [ expresión-constante_opt ]
        declarador-directo ( lista-tipos-de-parámetro )
        declarador-directo ( lista-de-identificadores_opt )
apuntador:
        * lista-calificadores-de-tipo_opt
        * lista-calificadores-de-tipo_opt apuntador
lista-calificadores-de-tipo:
        calificador-de-tipo
        lista-calificadores-de-tipo calificador-de-tipo
lista-tipos-de-parámetro:
        lista-de-parámetros
        lista-de-parámetros , . . .
lista-de-parámetros:
        declaración-parámetro
        lista-de-parámetros , declaración-parámetro
declaración-parámetro:
        especificadores-de-declaración declarador
        especificadores-de-declaración declarador-abstracto_opt
lista-de-identificadores:
        identificador
```



```
        lista-de-identificadores , identificador
inicializador:
        expresión-asignación
        { lista-de-inicializadores }
        { lista-de-inicializadores , }
lista-de-inicializadores:
        inicializador
        lista-de-inicializadores , inicializador
nombre-de-tipo:
        lista-calificador-especificador declarador-abstracto_opt
declarador-abstracto:
        apuntador
        apuntador_opt declarador-abstracto-directo
declarador-abstracto-directo:
        ( declarador-abstracto )
        declarador-abstracto-directo_opt [ expresión-constante_opt ]
        declarador-abstracto-directo_opt ( lista-tipos-de-
        parámetro_opt )
nombre-typedef:
        identificador
proposición:
        proposición-etiquetada
        proposición-expresión
        proposición-compuesta
        proposición-de-selección
        proposición-de-iteración
        proposición-de-salto
proposición-etiquetada:
        identificador : proposición
        case expresión-constante : proposición
        default : proposición
proposición-expresión:
        expresión_opt ;
proposición-compuesta:
        { lista-declaración_opt lista-de-proposiciones_opt }
lista-de-proposiciones:
        proposición
        lista-de-proposiciones proposición
proposición-de-selección:
        if ( expresión ) proposición
        if ( expresión ) proposición else proposición
        switch ( expresión ) proposición
proposición-de-iteración:
        while ( expresión ) proposición
        do proposición while ( expresión ) ;
        ( expresión_opt ; expresión_opt ; expresión_opt ) proposición
proposición-de-salto:
        goto identificador ;
        continue ;
```

```
        break ;
        return expresión_opt ;
expresión:
        expresión-de-asignación
        expresión , expresión-de-asignación
expresión-de-asignación:
        expresión-condicional
        expresión-unaria operador-de-asignación expresión-de-asignación
operador-de-asignación: uno de
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
expresión-condicional:
        expresión-lógica-OR
        expresión-lógica-OR ? expresión: expresión-condicional
expresión-constante:
        expresión-condicional
expresión-lógica-OR:
        expresión-lógica-AND
        expresión-lógica-OR || expresión-lógica-AND
expresión-lógica-AND:
        expresión-OR-inclusivo
        expresión-lógica-AND && expresión-OR-inclusivo
expresión-OR-inclusivo:
        expresión-OR-exclusivo
        expresión-OR-inclusivo | expresión-OR-exclusivo
expresión-OR-exclusivo:
        expresión-AND
        expresión-OR-exclusivo ^ expresión-AND
expresión-AND:
        expresión-de-igualdad
        expresión-AND & expresión-de-igualdad
expresión-de-igualdad:
        expresión-relacional
        expresión-de-igualdad == expresión-relacional
        expresión-de-igualdad != expresión-relacional
expresión-relacional:
        expresión-de-corrimiento
        expresión-relacional < expresión-de-corrimiento
        expresión-relacional > expresión-de-corrimiento
        expresión-relacional <= expresión-de-corrimiento
        expresión-relacional >= expresión-de-corrimiento
expresión-de-corrimiento:
        expresión-aditiva
        expresión-de-corrimiento << expresión-aditiva
        expresión-de-corrimiento >> expresión-aditiva
expresión-aditiva:
        expresión-multiplicativa
        expresión-aditiva + expresión-multiplicativa
        expresión-aditiva - expresión-multiplicativa
expresión-multiplicativa:
```



```
        expresión-cast
        expresión-multiplicativa * expresión-cast
        expresión-multiplicativa / expresión-cast
        expresión multiplicativa % expresión-cast
        expresión-cast:
        expresión-unaria
        ( nombre-de-tipo ) expresión-cast
expresión-unaria:
        expresión-posfija
        ++ expresión-unaria
        -- expresión-unaria
        operador-unario expresión-cast
        sizeof expresión-unaria
        sizeof ( nombre-de-tipo )
operador-unario: uno de
        &   *   +   -   ~   !
expresión-posfija:
        expresión-primaria
        expresión-posfija [ expresión ]
        expresión-posfija ( lista-de-expresiones-argumento_opt)
        expresión-posfija . identificador
        expresión-posfija -> identificador
        expresión-posfija ++
        expresión-posfija --
expresión-primaria:
        identificador
        constante
        cadena
        ( expresión )
lista-expresiones-argumento:
        expresión-de-asignación
        lista-expresiones-argumento , expresión-de-asignación
constante:
        constante-entera
        constante-de-carácter
        constante-flotante
        constante-enumeración
```

La siguiente gramática para el preprocesador resume la estructura de las líneas de control, pero no es adecuada para un *parser* mecanizado. Incluye el símbolo *text*, que es texto ordinario de programa, líneas de control no condicionales del preprocesador, o construcciones condicionales completas del preprocesador.

```
línea de control:
        # define identificador secuencia-de-tokens
        # define identificador ( identificador , ... , identificador ) secuencia-de-tokens
        # undef identificador
        # include <nombre-de-archivo>
        # include "nombre-de-archivo"
        # include secuencia-de-tokens
```

line *constante "nombre-de-archivo"*
line *constante*
error *secuencia-de-tokens$_{opt}$*
pragma *secuencia-de-tokens$_{opt}$*
#
preprocesador-condicional

preprocesador-condicional:
línea-if texto partes-elif parte-else$_{opt}$ # **endif**

línea-if:
if *expresión-constante*
ifdef *identificador*
ifndef *identificador*

partes-elif:
línea-elif texto
partes-elif$_{opt}$

línea-elif:
elif *expresión-constante*

parte-else:
línea-else texto

línea-else:
else

APENDICE B: **Biblioteca estándar**

Este apéndice es un resumen de la biblioteca definida por el estándar ANSI. La biblioteca estándar no es propiamente parte del lenguaje C, pero un entorno que opere con C estándar proporcionará las declaraciones y tipos de funciones y las macrodefiniciones de esta biblioteca. Hemos omitido algunas funciones que son de utilidad limitada o fácilmente sintetizadas a partir de otras; también hemos omitido caracteres de bytes múltiples así como la discusión de cuestiones locales, esto es, propiedades que dependen del lenguaje local, la nacionalidad o la cultura.

Las funciones, tipos y macros de la biblioteca estándar están declarados en *headers* estándar:

```
<assert.h>  <float.h>   <math.h>    <stdarg.h>  <stdlib.h>
<ctype.h>   <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>   <locale.h>  <signal.h>  <stdio.h>   <time.h>
```

Se puede tener acceso a un *header* por medio de

```
#include <header>
```

Los *headers* se pueden incluir en cualquier orden y número de veces. un *header* se debe incluir fuera de cualquier declaración o definición externa y antes de cualquier uso de cualquier cosa que declare. No es necesario que un *header* sea un archivo fuente.

Los identificadores externos que principian con subguión están reservados para uso de la biblioteca, como lo están todos los otros identificadores que principian con un subguión y una letra mayúscula u otro subguión.

B1. Entrada y salida: <stdio.h>

Las funciones, tipos y macros de entrada y salida, definidos en <stdio.h>, representan cerca de la tercera parte de la biblioteca.

Un *flujo* (*stream*) es una fuente o destino de datos que puede estar asociada con un disco u otro periférico. La biblioteca maneja flujos de texto y flujos binarios, aunque en algunos sistemas, notablemente UNIX, son idénticos. Un flujo de texto es una secuencia de líneas; cada línea tiene cero o más caracteres y está terminada por '\n'. Un entorno puede necesitar convertir un flujo de texto a alguna representación, o de alguna otra representación (tal como la asociación de '\n' a retorno de carro y avance de línea). Un flujo binario es una secuencia de bytes no procesados que representan datos internos, con la pro-

piedad de que si es escrito y después leído de nuevo en el mismo sistema, será comparado como igual.

Un flujo se conecta a un archivo o dispositivo al *abrirlo*; la conexión se rompe *cerrando* el flujo. El abrir un archivo regresa un apuntador a un objeto de tipo **FILE**, que registra cualquier información necesaria para controlar el flujo. Usaremos "apuntador de archivo" y "flujo" indistintamente cuando no haya ambigüedad.

Cuando un programa principia su ejecución, los flujos **stdin**, **stdout**, y **stderr** ya están abiertos.

B1.1 Operaciones sobre archivos

Las siguientes funciones tratan con operaciones sobre archivos. El tipo **size_t** es el tipo entero sin signo producido por el operador **sizeof**.

`FILE *fopen(const char *filename, const char *mode)`

fopen abre el archivo nombrado, y regresa un flujo, o **NULL** si falla el intento. Los valores legítimos de **mode** incluyen

`"r"`	abre archivo de texto para lectura
`"w"`	crea archivo de texto para escritura; descarta el contenido previo si existe
`"a"`	agrega; abre o crea un archivo para escribir al final
`"r+"`	abre archivo para actualización (esto es, lectura o escritura)
`"w+"`	crea archivo de texto para actualización; descarta cualquier contenido previo si existe
`"a+"`	agrega; abre o crea archivo de texto para actualización, escribiendo al final

El modo de actualización permite la lectura y escritura del mismo archivo; entre una lectura y una escritura debe llamarse a **fflush** o a una función de posición o viceversa. Si el modo incluye **b** después de la letra inicial, como en `"rb"` or `"w+b"`, indica que el archivo es binario. Los nombres de archivo están limitados a **FILENAME_MAX** caracteres. Cuando más pueden ser abiertos **FOPEN_MAX** archivos a la vez.

`FILE *freopen(const char *filename, const char *mode, FILE *stream)`

freopen abre el archivo con el modo estipulado y asocia al flujo con él. Regresa **stream**, o **NULL** si ocurre un error. **freopen** normalmete se usa para cambiar los archivos asociados con **stdin**, **stdout** o **stderr**.

`int fflush(FILE *stream)`

Para un flujo de salida, **fflush** ocasiona que sea escrito cualquier dato con uso de buffer que hasta ese momento no hay sido escrito; para un flujo de entrada, el efecto está indefinido. Regresa **EOF** en caso de un error de escritura, y cero en caso contrario.

`int fclose(FILE *stream)`

fclose descarga cualquier dato no escrito de **stream**, descarta cualquier buffer de entrada no leído, libera cualquier buffer asignado automáticamente, después cierra el flujo. Regresa **EOF** si ocurre cualquier error y cero en caso contrario.

`int remove(const char *filename)`

remove remueve el archivo nombrado, de modo que un intento posterior de abrirlo fallará. Regresa un valor diferente de cero si el intento falla.

`int rename(const char *oldname, const char *newname)`

rename cambia el nombre de un archivo; regresa un valor diferente de cero si el intento falla.

`FILE *tmpfile(void)`

tmpfile crea un archivo temporal con modo `"wb+"` que será removido automáticamente cuando se cierre o cuando el programa termine normalmente. **tmpfile** regresa un flujo, o **NULL** si no puede crear el archivo.

`char *tmpnam(char s[L_tmpnam])`

tmpnam(NULL) crea una cadena que no es el nombre de un archivo existente y regresa un apuntador a un arreglo estático interno. **tmpname(s)** almacena la cadena en s y también la regresa como el valor de la función; s debe tener espacio suficiente para al menos **L_tmpnam** caracteres. **tmpnam** genera un nombre diferente cada vez que se invoca; cuando más están garantizados `TMP_MAX` diferentes nombres durante la ejecución del programa. Nótese que **tmpnam** crea un nombre, no un archivo.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`

setvbuf controla el uso de buffer para el flujo; se debe invocar antes de leer o escribir. Un modo `_IOFBF` ocasiona uso completo de buffers, `_IOLBF` uso de buffers por línea de archivo de texto, e `_IONBF` ningún uso de buffer. Si **buf** no es **NULL**, se empleará como el buffer, de otra manera será asignado un buffer. **size** determina su tamaño. **setvbuf** regresa un valor diferente de cero en caso de cualquier error.

`void setbuf(FILE *stream, char *buf)`

Si **buf** es **NULL**, se suspende el uso de buffer para el flujo. De otra manera, **setbuf** es equivalente a `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

B1.2 Salida con formato

Las funciones **printf** proporcionan conversiones de salida con formato.

`int fprintf(FILE *stream, const char *format, ...)`

fprintf toma una salida y la convierte y escribe hacia el **stream** bajo el control de **format**. El valor regresado es el número de caracteres escritos, o un valor negativo si ocurrió algún error.

La cadena de formato contiene dos tipos de objetos: caracteres ordinarios, que son copiados al flujo de salida, y especificaciones de conversión, cada uno de los cuales provoca la conversión e impresión de los siguientes argumentos sucesivos de **fprintf**. Cada especificación de conversión principia con el carácter **%** y termina con un carácter de conversión. Entre el % y el carácter de conversión puede haber, en orden:

- Banderas (en cualquier orden), que modifican la especificación:
  - **-**, que especifica ajuste del argumento convertido hacia la izquierda dentro de su campo.
  - **+**, que estipula que el número siempre será impreso con signo.
  - *espacio*: si el primer carácter no es un signo, se prefijará un espacio.
  - **0**: para conversión numérica, estipula rellenado con ceros iniciales de la totalidad del campo.
  - **#**, que estipula una forma alterna de salida. Para **o**, el primer dígito será cero. Para **x** o **X**, cual-

quier resultado diferente de cero será prefijado con 0x o 0X. Para e, E, f, g, y G, la salida siempre tendrá un punto decimal; para g y G, los ceros acarreados no serán removidos.

- Un número que estipula un ancho mínimo de campo. El argumento convertido será impreso en un campo de por lo menos esta amplitud, y en una mayor si es necesario. Si el argumento convertido tiene menos caracteres que el ancho de campo, será rellenado a la izquierda (o derecha, si se ha requerido justificación a la izquierda) para completar el ancho de campo. El carácter de relleno normalmente es espacio, pero es 0 si está presente la bandera de relleno con ceros.
- Un punto, que separa el ancho de campo de la precisión.
- Un número, la precisión, que estipula el número máximo de caracteres de una cadena que serán impresos, o el número de dígitos que serán impresos después del punto decimal para conversiones e, E, o f, o el número de dígitos significativos para conversiones g o G, o el número mínimo de dígitos que serán impresos para un entero (serán agregados ceros al principio para completar el ancho de campo necesario).
- Un modificador de longitud h, l (letra ele), o L. "h" indica que el argumento correspondiente va a ser impreso como short o unsigned short; "l" indica que el argumento es long o unsigned long; "L" indica que el argumento es long double.

Con * se puede especificar la amplitud o precisión, o ambas, en tal caso el valor se calcula convirtiendo el (los) siguiente(s) argumento(s), que debe(n) ser int.

Los caracteres de conversión y sus significados se muestran en la tabla B-1. Si el carácter que está después del % no es un carácter de conversión, el comportamiento está indefinido.

TABLA B-1. CONVERSIONES DE PRINTF

CARACTER	TIPO DE ARGUMENTO; CONVERTIDO A
d, i	int; notación decimal con signo.
o	int; notación octal sin signo (sin ceros al principio)
x,X	int; notación hexadecimal sin signo (sin 0x o 0X al principio), utilizando abcdef para 0x o ABCDEF para 0X.
u	int; notación decimal sin signo.
c	int; carácter sencillo, después de la conversión a unsigned char.
s	char *; los caracteres de la cadena son impresos hasta que se alcanza un '\0' o hasta que ha sido impreso el número de caracteres indicados por la precisión.
f	double; notación decimal de la forma [−]*mmm.ddd*, en donde el número de *d* es especificado por la precisión. La precisión por omisión es 6; una precisión 0 suprime el punto decimal.
e,E	double; notación decimal de la forma [−]*m.dddddde±xx* o [−]*m.ddddddE±xx*, en donde el número de *d* está especificado por la precisión. La precisión por omisión es 6; una precisión 0 suprime el punto decimal.
g,G	double; es usa %e o %E si el exponente es menor que −4 o mayor o igual que la precisión; de otra forma es usado %f. Los ceros y el punto decimal al final no son impresos.
p	void *; imprime como un apuntador (representación dependiente de la implantación).
n	int *; el número de caracteres escritos hasta el momento por esta llamada a printf es *escrito en* el argumento. No es convertido ningún argumento.
%	No es convertido ningún argumento; se imprime como %.


```
int printf(const char *format, ...)
```

printf(...) es equivalente a fprintf(stdout,...).

```
int sprintf(char *s, const char *format, ...)
```

sprintf es lo mismo que printf excepto que la salida es escrita en la cadena s, terminada con '\0'. s debe ser suficientemente grande para contener el resultado. La cuenta regresada no incluye el '\0'.

```
vprintf(const char *format, va_list arg)
vfprintf(FILE *stream, const char *format, va_list arg)
vsprintf(char *s, const char *format, va_list arg)
```

Las funciones vprintf, vfprintf, y vsprintf son equivalentes a las correspondientes funciones printf, excepto que la lista variable de argumentos es remplazada por arg, que ha sido inicializado por la macro va_start y tal vez llamadas a va_arg. Véase la exposición de <stdarg.h> en la sección B7.

B1.3 Entrada con formato

Las funciones scanf tratan con la conversión de entrada con formato.

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf lee del stream bajo el control de format, y asigna los valores convertidos a través de argumentos subsecuentes, *cada uno de los cuales debe ser un apuntador*. Regresa cuando format se ha agotado. fscanf regresa EOF si se encuentra fin de archivo o un error antes de la conversión; de otra forma regresa el número de artículos de entrada convertidos y asignados.

La cadena de formato generalmente contiene especificaciones de conversión, que son utilizadas para dirigir la interpretación de la entrada. La cadena de formato puede contener:

- Blancos o tabuladores, que son ignorados.
- Caracteres ordinarios (no %), que se espera coincidan con los siguientes caracteres que no son espacio en blanco del flujo de entrada.
- Especificaciones de conversión, consistentes en %, un carácter optativo de supresión de asignación *, un número optativo que especifica una amplitud máxima de campo, una h, l o L optativa que indica la amplitud del objetivo, y un carácter de conversión.

Una especificación de conversión determina la conversión del siguiente campo de entrada. Normalmente el resultado es situado en la variable apuntada por el argumento correspondiente. Sin embargo, si se indica supresión de asignación con *, como en %*s, el campo de entrada simplemente se salta; no se hace asignación. Un campo de entrada está definido por una cadena de caracteres diferentes de espacio en blanco; se extiende hasta el siguiente carácter de espacio en blanco o hasta que el ancho de campo, si está especificado, se ha agotado. Esto implica que scanf leerá más allá de los límites de la línea para encontrar su entrada, ya que las nuevas líneas son espacios en blanco. (Los caracteres de espacio en blanco son blanco, tabulador, nueva línea, retorno de carro, tabulador vertical y avance de línea.)

El carácter de conversión indica la interpretación del campo de entrada. El argumento correspondiente debe ser un apuntador. Los caracteres de conversión legales se muestran en la tabla B-2.

Los caracteres de conversión d, i, n, o, u, y x pueden estar precedidos por h si el argumento es un apuntador a short en vez de int, o por l (letra ele) si el argumento es un apuntador a long. Los caracteres de conversión e, f, y g pueden estar precedidos por l si en

la lista de argumentos hay un apuntador a **double** y no a **float**, y por **L** si hay un apuntador a long double.

TABLA B-2. CONVERSIONES DE SCANF

CARACTER	DATO DE ENTRADA; TIPO DE ARGUMENTO
d	entero decimal; int *.
i	entero; int *. El entero puede estar en octal (iniciado con 0) o hexadecimal (iniciado con 0x o 0X).
o	entero octal (con o sin cero al principio); int *.
u	entero decimal sin signo; unsigned int *.
x	entero hexadecimal (con o sin 0x o 0X al principio); int *.
c	caracteres; char *. Los siguientes caracteres de entrada se pondrán en el arreglo indicado, hasta el número dado por el ancho de campo; el valor por omisión es 1. No se agrega '\0'. En este caso se suprime el salto normal sobre los caracteres de espacio en blanco; para leer el siguiente carácter que no sea blanco, use %1s.
s	cadena de caracteres que no es espacio en blanco (no entrecomillados); char *, apunta a un arreglo de caracteres suficientemente grande para contener la cadena y un '\0' terminal que se le agregará.
e,f,g	número de punto flotante; float *. El formato de entrada para los float es un signo optativo, una cadena de números posiblemente con un punto decimal, y un campo optativo de exponente con una E o e seguida posiblemente de un entero con signo.
p	valor apuntador como se imprime por printf ("%p"); void *.
n	escribe en el argumento el número de caracteres escritos hasta el momento por esta llamada; int *. No se lee entrada alguna. La cuenta de elementos convertidos no se incrementa.
[...]	coincide con la mayor cadena no vacía de caracteres de entrada del conjunto entre corchetes; char *. Se agrega un '\0'. [ ]...] incluye] en el conjunto.
[^...]	coincide con la mayor cadena no vacía de caracteres de entrada que *no* sean del conjunto entre corchetes; char *. Se agrega un '\0'. [^]...] incluye] en el conjunto.
%	literal; no se hace ninguna asignación.

`int scanf(const char *format, ...)`
scanf(...) es idéntica a fscanf(stdin,...).

`int sscanf(char *s, const char *format, ...)`
sscanf(s,...) es equivalente a scanf(...) excepto que los caracteres de entrada son tomados de la cadena s.

B1.4 Funciones de entrada y salida de caracteres

`int fgetc(FILE *stream)`
fgetc regresa el siguiente carácter de stream como unsigned char (convertido a un int), o EOF si se encontró el fin de archivo o un error.

`char *fgets(char *s, int n, FILE *stream)`
fgets lee hasta los siguientes n−1 caracteres en el arreglo s, deteniéndose si encuentra nueva línea; el nueva línea es incluido en el arreglo, que es terminado por '\0'. fgets regresa s, o NULL si se encuentra fin de archivo u ocurre un error.

`int fputc(int c, FILE *stream)`
fputc escribe el carácter c (convertido a unsigned char) en stream. Regresa el carácter escrito, o EOF en caso de error.

`int fputs(const char *s, FILE *stream)`
fputs escribe la cadena s (que no necesita contener '\n') en stream; regresa un valor no negativo, o EOF si hay error.

`int getc(FILE *stream)`
getc es equivalente a fgetc excepto que si es una macro, puede evaluar a stream más de una vez.

`int getchar(void)`
getchar es equivalente a getc(stdin).

`char *gets(char *s)`
gets lee la siguiente línea de entrada y la deja en el arreglo s; reemplaza el carácter nueva línea final con '\0'. Regresa s, o NULL si ocurre fin de archivo o error.

`int putc(int c, FILE *stream)`
putc es equivalente a fputc excepto que, si es una macro, puede evaluar a stream más de una vez.

`int putchar(int c)`
putchar(c) es equivalente a putc(c,stdout).

`int puts(const char *s)`
puts escribe la cadena s y un nueva línea a stdout. Regresa EOF si ocurre un error, de otra forma, un valor no negativo.

`int ungetc(int c, FILE *stream)`
ungetc regresa c (convertido en unsigned char) de nuevo al stream, de donde será regresado en la próxima lectura. Sólo se garantiza un carácter de regreso por flujo. EOF no puede ser regresado. ungetc devuelve el carácter regresado, o EOF en caso de error.

B1.5 Funciones de entrada y salida directa

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`
fread lee de stream en el arreglo ptr hasta nobj objetos de tamaño size. fread regresa el número de objetos leídos; esto puede ser menor que el número solicitado. Para determinar el status deben utilizarse feot y ferror.

```
size_t fwrite(const void *ptr, size_t size, size_t nobj,
                        FILE *stream)
```
fwrite escribe, del arreglo ptr, nobj objetos de tamaños size en stream. Devuelve el número de objetos escritos, que es menor que nobj en caso de error.

B1.6 Funciones de colocación en archivos

`int fseek(FILE *stream, long offset, int origin)`

fseek fija la posición en el archivo para el stream; una lectura o escritura posterior tendrá acceso a datos que principian en la nueva posición. Para un archivo binario, la posición se fija a **offset** caracteres de **origin**, el cual puede ser **SEEK_SET** (principio), **SEEK_CUR** (posición actual), o **SEEK_END** (fin del archivo). Para un **stream** de texto, **offset** debe ser cero, o un valor regresado por **ftell** (en tal caso **origin** debe ser **SEEK_SET**.) fseek regresa un valor diferente de cero en caso de error.

`long ftell(FILE *stream)`

ftell regresa la posición actual de **stream** en el archivo, o −1L en caso de error.

`void rewind(FILE *stream)`

rewind(fp) es equivalente a fseek(fp,0L,SEEK_SET); clearerr(fp).

`int fgetpos(FILE *stream, fpos_t *ptr)`

fgetpos graba en *ptr la posición actual de stream, para uso posterior de fsetpos. El tipo fpos_t es adecuado para grabar tales valores. fgetpos regresa un valor diferente de cero en caso de error.

`int fsetpos(FILE *stream, const fpos_t *ptr)`

fsetpos coloca **stream** en la posición grabada en *ptr por **fgetpos**. En caso de error, fsetpos regresa un valor diferente de cero.

B1.7 Funciones de error

Muchas funciones de la biblioteca activan indicadores de estado cuando ocurre un error o fin de archivo. Estos indicadores se pueden fijar y probar explícitamente. Además, la expresión entera errno (declarada en <errno.h> puede contener un número de error que da información adicional acerca del error más reciente.

`void clearerr(FILE *stream)`

clearrer limpia los indicadores de fin de archivo y error para el **stream**.

`int feof(FILE *stream)`

feof regresa un valor diferente de cero si está encendido el indicador de fin de archivo para stream.

`int ferror(FILE *stream)`

ferror regresa un valor diferente de cero si está encendido el indicador de error de stream.

`void perror(const char *s)`

perror(s) imprime s y un mensaje de error definido por la implantación, correspondiente al entero que está en errno, como si fuera

`fprintf(stderr, "%s: %s\n", s, "`*mensaje de error*`")`

Ver strerror en la sección B3.

B2. Pruebas de clasificación de caracteres: <ctype.h>

El *header* <ctype.h> declara funciones para la prueba de caracteres. Para cada función, el argumento es un int cuyo valor debe ser EOF o representable por un **unsigned char**, y el valor de retorno es un int. Las funciones regresan diferente de cero (verdadero) si el argumento c satisface la condición descrita, y cero si no lo hace.

`isalnum(c)`	`isalpha(c)` o `isdigit(c)` es verdadera
`isalpha(c)`	`isupper(c)` o `islower(c)` es verdadera
`iscntrl(c)`	carácter de control
`isdigit(c)`	dígito decimal
`isgraph(c)`	carácter de impresión excepto espacio
`islower(c)`	letra minúscula
`isprint(c)`	carácter de impresión incluyendo espacio
`ispunct(c)`	carácter de impresión excepto espacio, letra o dígito.
`isspace(c)`	espacio, avance de línea, nueva línea, retorno de carro, tabulador, tabulador vertical
`isupper(c)`	letra mayúscula
`isxdigit(c)`	dígito hexadecimal

En el conjunto de caracteres ASCII de siete bits, los caracteres de impresión son de 0x20 (' ') a 0x7E ('~'); os caracteres de control son de 0 (NUL) a 0x1F (US) y 0x7F (DEL).

Además, hay dos funciones que convierten letras:

`int tolower(int c)`	convierte c a minúscula
`int toupper(int c)`	convierte c a mayúscula

Si c es una letra mayúscula, tolower(c) regresa la correspondiente letra minúscula; en otro caso regresa c. Si c es una letra minúscula, toupper(c) regresa la correspondiente letra mayúscula; en otro caso regresa c.

B3. Funciones para cadenas: <string.h>

Hay dos grupos de funciones para cadenas definidas en el *header* <string.h>. Las primeras tienen nombres que comienzan con **str**; las segundas tienen nombres que comienzan con **mem**. Excepto para **memmove**, el comportamiento está indefinido si la copia ocurre entre objetos que se traslapan.

En la siguiente tabla, las variables s y t son de tipo char *; cs y ct son de tipo **const char** *; n es de tipo **size_t**; y c es un int convertido a **char**

`char *strcpy(s,ct)`	copia la cadena ct a la cadena s, incluyendo '\0'; regresa s.
`char *strncpy(s,ct,n)`	copia hasta n caracteres de la cadena ct a s; regresa s. Rellena con '\0' si ct tiene menos de n caracteres.
`char *strcat(s,ct)`	concatena la cadena ct al final de la cadena s; regresa s;
`char *strncat(s,ct,n)`	concatena hasta n caracteres de la cadena ct a la cadena s, terminando con '\0'; regresa s.
`int strcmp(cs,ct)`	compara la cadena cs con la cadena ct; regresa <0 si cs<ct, 0 si cs==ct, o >0 if cs>ct.
`int strncmp(cs,ct,n)`	compara hasta n caracteres de la cadena cs con la cadena ct; regresa <0 if cs<ct, 0 si cs==ct, o >0 si cs>ct.
`char *strchr(cs,c)`	regresa un apuntador a la primera ocurrencia de c en cs, o NULL si no está presente.
`char *strrchr(cs,c)`	regresa un apuntador a la última ocurrencia de c en cs, o NULL si no está presente.

`size_t strspn(cs,ct)`	regresa la longitud del prefijo de cs que consiste en los caracteres en ct.
`size_t strcspn(cs,ct)`	regresa la longitud del prefijo de cs que consiste en los caracteres *que no* están en ct.
`char *strpbrk(cs,ct)`	regresa un apuntador a la primera ocurrencia en la cadena cs de cualquier carácter de la cadena ct, o NULL si ninguno está presente.
`char *strstr(cs,ct)`	regresa un apuntador a la primera ocurrencia de la cadena ct en cs, o NULL si no está presente.
`size_t strlen(cs)`	regresa la longitud de cs.
`char *strerror(n)`	regresa un apuntador a una cadena definida por la implantación, correspondiente al error n.
`char *strtok(s,ct)`	**strtok** busca en s *tokens* delimitados por caracteres de ct; ver abajo.

Una secuencia de llamadas a **strtok(s,ct)** divide s en *tokens*, cada uno delimitado por un carácter de **ct**. **La primer llamada** en la secuencia do tiene una s no **NULL**. Encuentra el primer *token* en s que consiste en caracteres que no están **en ct**; termina al sobreescribir el siguiente carácter **de** s con '\0' y regresa un apuntador al *token*. Cada llamada subsiguiente, indicada por un valor **NULL** de **s**, regresa el *token* siguiente, buscando justo a partir del final del anterior. **strtok** regresa **NULL** cuando ya no encuentra *tokens*. La cadena **ct** puede ser diferente en cada llamada.

La intención de las funciones mem... es manipular objetos como arreglos de caracteres; la intención es lograr una interfaz para rutinas eficientes. En la siguiente tabla, **s y t** son de tipo void *; cs y ct son de tipo **const void *; n es de tipo size_t; y c es un int** convertido a **unsigned char**.

`void *memcpy(s,ct,n)`	copia n caracteres de ct a s, y regresa s.
`void *memmove(s,ct,n)`	lo mismo que memcpy excepto que funciona aun si los objetos se traslapan.
`int memcmp(cs,ct,n)`	compara los primeros n caracteres de cs con ct; regresa lo mismo que strcmp.
`void *memchr(cs,c,n)`	regresa un apuntador a la primera ocurrencia del carácter c en cs, o NULL si no está presente entre los primeros n caracteres.
`void *memset(s,c,n)`	coloca el caracter c en los primeros n caracteres de s, regresa s.

B4. Funciones matemáticas: <math.h>

El header <math.h> declara funciones y macros matemáticas.

Las macros **EDOM** y **ERANGE** (que se encuentran en **<errno.h>**) son constantes enteras con valor diferente de cero que se usan para señalar errores de dominio y de rango para las funciones; **HUGE_VAL** es un valor positivo **double**. Un *error de dominio* ocurre si un argumento está fuera del dominio sobre el que está definida la función. En caso de un error de dominio, **errno** toma el valor de **EDOM**; el valor regresado depende de la implantación. Un *error de rango* ocurre si el resultado de la función no se puede representar como

un **double**. Si el resultado se desborda, la función regresa **HUGE_VAL** con el signo correcto, y **errno** se hace **ERANGE**. Si el resultado es tan pequeño que no se puede representar (**underflow**), la función regresa cero; el que **errno** sea fijado a **ERRANGE** depende de la implantación.

En la tabla siguiente, **x** y **y** son de tipo **double**, **n** es un **int**, y todas las funciones regresan **double**. Los ángulos para las funciones trigonométricas están representados en radianes.

`sin(x)`	seno de x
`cos(x)`	coseno de x
`tan(x)`	tangente de x
`asin(x)`	$\sin^{-1}(x)$ en el rango $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.
`acos(x)`	$\cos^{-1}(x)$ en el rango $[0, \pi]$, $x \in [-1, 1]$.
`atan(x)`	$\tan^{-1}(x)$ en el rango $[-\pi/2, \pi/2]$.
`atan2(y,x)`	$\tan^{-1}(y/x)$ en el rango $[-\pi, \pi]$.
`sinh(x)`	seno hiperbólico de x
`cosh(x)`	coseno hiperbólico de x
`tanh(x)`	tangente hiperbólica de x
`exp(x)`	función exponencial e^x
`log(x)`	logaritmo natural $\ln(x)$, $x>0$.
`log10(x)`	logaritmo base 10 $\log_{10}(x)$, $x>0$.
`pow(x,y)`	x^y. *Ocurre un error de dominio si* **x**=0 y y≤0, o si **x**<0 y y no es un entero.
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$.
`ceil(x)`	menor entero no menor que x, como double.
`floor(x)`	mayor entero no mayor que x, como double.
`fabs(x)`	valor absoluto $\lvert x \rvert$
`ldexp(x,n)`	$x \cdot 2^n$
`frexp(x, int *exp)`	divide **x** en una fracción normalizada dentro del intervalo [1/2, 1], que se regresa, y una potencia de 2, que se almacena en ***exp**. Si **x** es cero, ambas partes del resultado son cero.
`modf(x, double *ip)`	divide **x** en parte entera y fraccionaria, cada una con el mismo signo que **x**. Almacena la parte entera en *ip, y regresa la parte fraccionaria.
`fmod(x,y)`	residuo de punto flotante de **x/y**, con el mismo signo que **x**. Si y es cero, el resultado está definido por la implantación.

B5. Funciones de utilería: <stdlib.h>

El *header* <stdlib.h> declara funciones para conversión numérica, asignación de memoria y tareas semejantes.

`double atof(const char *s)`

atof convierte s a double; es equivalente a `strtod(s, (char**)NULL)`.

`int atoi(const char *s)`
convierte s a int; es equivalente a `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char *s)`
convierte s a long; es equivalente a `strtol(s, (char**)NULL, 10)`.

`double strtod(const char *s, char **endp)`
strtod convierte el prefijo de s a **double**, ignorando el espacio en blanco inicial; almacena en ***endp** un apuntador a cualquier sufijo no cubierto salvo cuando **endp** es **NULL**. Si la respuesta se desborda, regresa **HUGE_VAL** con el signo apropiado; si el resultado fuera tan pequeño que no se pueda representar (*underflow*), se regresa cero. En cualquier caso **errno** toma el valor **ERANGE**.

`long strtol(const char *s, char **endp, int base)`
strtol convierte el prefijo de s a long, ignorando los espacios en blanco iniciales; almacena un apuntador en ***endp** a cualquier sufijo no cubierto a menos de que **endp** sea **NULL**. Si base está entre 2 y 36, la conversión se realiza suponiendo que la entrada es escrita en esa base. Si base es cero, la base es 8, 10, o 16; los **0** iniciales implican octal, mientras que **0x** y **0X**, hexadecimal. Las letras, ya sean mayúsculas o minúsculas, representan dígitos desde 10 hasta base—1; en base 16 se permite iniciar con **0x** o **0X**. Si el resultado se desborda, se regresa **LONG_MAX** o **LONG_MIN**, dependiendo del signo del resultado, y **errno** se hace **ERANGE**.

`unsigned long strtoul(const char *s, char **endp, int base)`
strtoul es lo mismo que strtol excepto que el resultado es unsigned long y el valor de error es **ULONG_MAX**.

`int rand(void)`
rand devuelve un entero pseudoaleatorio en el rango de 0 a **RAND_MAX**, que es al menos 32767.

`void srand(unsigned int seed)`
srand utiliza **seed** como semilla para una nueva secuencia de números pseudoaleatorios. La semilla inicial es 1.

`void *calloc(size_t nobj, size_t size)`
calloc devuelve un apuntador al espacio para un arreglo de **nobj** objetos, cada uno de tamaño size, o NULL si la solicitud no puede satisfacerse. El espacio se inicializa a bytes con cero.

`void *malloc(size_t size)`
malloc regresa un apuntador al espacio para un objeto de tamaño **size**, o **NULL** si la solicitud no se puede satisfacer. El espacio no se inicializa.

`void *realloc(void *p, size_t size)`
realloc cambia el tamaño del objeto apuntado por **p** a **size**. El contenido no cambiará sino hasta el mínimo de los tamaños viejo y nuevo. Si el nuevo tamaño es mayor, el espacio nuevo no se inicializa. **realloc** regresa un apuntador al nuevo espacio, o **NULL** si la solicitud no se puede satisfacer; en tal caso ***p** no cambia.

`void free(void *p)`
free desasigna el espacio apuntado por p; si p es **NULL**, no hace nada. p debe ser un apuntador a un espacio previamente asignado por **calloc**, **malloc**, o **realloc**.

`void abort(void)`
abort ocasiona que el programa termine anormalmente, como con **raise(SIGABRT)**.

`void exit(int status)`
exit ocasiona la terminación normal del programa. Las funciones **atexit** se llaman en orden inverso del registrado, los archivos abiertos se vacían, los flujos abiertos se cierran, y el control se regresa al entorno. Cómo se regrese **status** al entorno depende de la implantación, pero cero indica cuando la terminación tiene éxito. Se pueden utilizar también los valores **EXIT_SUCCESS** y **EXIT_FAILURE**

`int atexit(void (*fcn)(void))`
atexit registra a la función fcn para que sea llamada cuando el programa termina normalmente; regresa un valor diferente de cero cuando no se puede hacer el registro.

`int system(const char *s)`
system pasa la cadena s al entorno para que se ejecute. Si s es **NULL**, system regresa un valor diferente de cero si hay un procesador de órdenes. Si s no es **NULL**, el valor regresado depende de la implantación.

`char *getenv(const char *name)`
getenv regresa la cadena del entorno asociada con **name**, o **NULL** si no existe. Los detalles dependen de la implantación.

```
void *bsearch(const void *key, const void *base,
      size_t n, size_t size,
      int (*cmp)(const void *keyval, const void *datum))
```
bsearch busca en base[0]...base[n−1] un elemento que coincida con *key. La función **cmp** debe regresar un valor negativo si su primer argumento (la llave de búsqueda) es menor que su segundo (una entrada en la tabla), cero si es igual, y positivo para mayor. Los elementos del arreglo base deben estar en orden ascendente. **bsearch** regresa un apuntador al elemento coincidente, o **NULL** si no existe.

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *))
```
qsort clasifica en orden ascendente un arreglo **base[0]...base[n−1]** de objetos de tamaño size. La función de comparación **cmp** es como en **bsearch**.

`int abs(int n)`
abs regresa el valor absoluto de su argumento int.

`long labs(long n)`
labs regresa el valor absoluto de su argumento long.

`div_t div(int num, int denom)`
div calcula el cociente y el residuo de **num/denom**. Los resultados se almacenan en los miembros de tipo int quot y rem de una estructura de tipo `div_t`.

`ldiv_t ldiv(long num, long denom)`
ldiv calcula el cociente y el residuo de **num/denom**. Los resultados son almacenados en los miembros de tipo **long quot** y **rem** de una estructura de tipo `ldiv_t`.

B6. Diagnósticos: <assert.h>

La macro **assert** es usada para agregar diagnósticos a los programas:

```
void assert(int expresión)
```

Si *expresión* es cero cuando

```
assert(expresión)
```

se ejecuta, la macro **assert** imprimirá en **stderr** un mensaje, como

```
Assertion failed: expresión, file filename, line nnn
```

Después llama a **abort** para terminar la ejecución. El archivo fuente **filename** y el número de línea viene de las macros `__FILE__` y `__LINE__` del preprocesador.

Si **NDEBUG** está definido cuando se incluyó `<assert.h>` se ignora la macro **assert**.

B7. Listas de argumentos variables: <stdarg.h>

El *header* <stdarg.h> proporciona recursos para recorrer una lista de argumentos de función de tamaño y tipo desconocido.

Supóngase que *lastarg* es el último parámetro nombrado de una función **f** con un número variable de argumentos. Entonces se declara dentro de **f** una variable **ap** de tipo **va_list** que apuntará a cada argumento en orden:

```
va_list ap;
```

ap se debe inicializar una vez con la macro **va_list** antes de tener acceso a cualquier argumento no nombrado:

```
va_start(va_list ap, lastarg);
```

Después de eso, cada ejecución de la macro **va_arg** producirá un valor que tiene el tipo y valor del siguiente argumento no nombrado, y modificará también **ap** de modo que el próximo uso de **va_arg** devuelva el argumento siguiente:

```
type va_arg(va_list ap, type);
```

La macro

```
void va_end(va_list ap);
```

se debe llamar una vez después de que han sido procesados los argumentos, pero antes de haber salido de **f**.

B8. Saltos no locales: <setjmp.h>

Las declaraciones que están en <setjmp.h> proporcionan una forma de evitar la secuencia normal de llamada y regreso de funciones, típicamente para permitir un regreso inmediato de una llamada a una función profundamente anidada.

`int setjmp(jmp_buf env)`

La macro **setjmp** guarda la información del estado que se tiene en **env** para ser usada por **longjmp**. El retorno es cero desde una llamada directa de **setjmp** y diferente de cero desde una llamada subsiguiente a **longjmp**. Sólo puede ocurrir una llamada a **setjmp** dentro de ciertos contextos, básicamente la prueba de **if**, **switch**, y ciclos, y sólo en expresiones de relación simples.

```
if (setjmp(env) == 0)
    /* llega aquí en una llamada directa */
else
    /* llega aquí por una llamada de longjmp */
```


`void longjmp(jmp_buf env, int val)`

longjmp restablece el estado guardado por la llamada más reciente de **setjmp**, utilizando la información almacenada en **env**; la ejecución continúa como si la función **setjmp** sólo hubiera sido llamada y hubiera regresado un valor de **val** diferente de cero. La función que contiene **setjmp** no debe haber terminado. Los objetos accesibles tienen los valores que tenían en el momento en que **longjmp** fue llamada; los valores no son guardados por **setjmp**.

B9. Señales: <signal.h>

El *header* `<signal.h>` de facilidades para manejar condiciones excepcionales que aparecen durante la ejecución, tal como una señal de interrupción de una fuente externa o un error en la ejecución.

`void (*signal(int sig, void (*handler)(int)))(int)`

signal determina cómo se manejarán las señales subsiguientes. Si **handler** es `SIG_DFL`, se usa el comportamiento predefinido por la implantación; si es `SIG_IGN`, a señal se ignora; de otra manera, se llamará a la función apuntada por **handler**, con los argumentos del tipo de la señal. Las señales válidas incluyen

`SIGABRT`	terminación anormal, p. ej., desde
`SIGFPE`	error aritmético, p. ej., división entre cero o sobreflujo
`SIGILL`	imagen de función ilegal, p. ej., instrucción ilegal
`SIGINT`	atención ilegal al almacenamiento; p. ej., acceso fuera de los límites
`SIGSEGV`	acceso ilegal al almacenamiento, vgr., acceso fuera de los límites de memoria
`SIGTERM`	solicitud de terminación enviada a este programa

signal regresa el valor previo de **handler** para la señal específica, o `SIG_ERR` si ocurre un error.

Cuando ocurre subsecuentemente una señal **sig**, la señal se regresa a su comportamiento predeterminado; luego se llama a la función manejadora de la señal, como si se ejecutara **(*handler)(sig)**. Si el manejador regresa, la ejecución continuará donde se encontraba cuando ocurrió la señal.

El estado inicial de las señales está definido por la implantación.

`int raise(int sig)`

raise envía la señal **sig** al programa; regresa un valor diferente de cero cuando no tiene éxito.

B10. Funciones de fecha y hora <time.h>

El *header* `<time.h>` declara los tipos y funciones para manipulación de fecha y hora. Algunas funciones procesan la *hora local*, que puede diferir de la del calendario, por ejemplo, debido a la zona horaria. **clock_t** y **time_t** son tipos aritméticos que representan tiempos, y **struct tm** mantiene las componentes de un calendario:

```
int tm_sec;      segundos después del minuto (0,59)
int tm_min;      minutos después de la hora (0,59)
int tm_hour;     horas desde la medianoche (0,23)
int tm_mday;     día del mes (1,31)
int tm_mon;      meses desde enero (0,11)
int tm_year;     años desde 1900
int tm_wday;     días desde el domingo (0,6)
int tm_yday;     días desde enero 1 (0,365)
int tm_isdst;    bandera de Daylight Saving Time
```

tm_isdst es positiva si *Daylight Saving Time* está en efecto, cero si no lo está y negativa si la información no está disponible.

`clock_t clock(void)`
clock regresa el tiempo de procesador empleado por el programa desde el inicio de su ejecución, o −1 si no está disponible. clock( )/CLK_TCK es el tiempo en segundos.

`time_t time(time_t *tp)`
time regresa la fecha y hora actual del calendario, o −1 si no está disponible. Si tp no es NULL, el valor de retorno también es asignado a *tp.

`double difftime(time_t time2, time_t time1)`
difftime regresa time2-time1 expresado en segundos.

`time_t mktime(struct tm *tp)`
mktime convierte la fecha y hora local de la estructura *tp a fecha y hora del calendario en la misma representación utilizada por time. Los componentes tendrán valores dentro de los rangos mostrados. mktime regresa la fecha y hora del calendario, o −1 si no puede ser representada.

Las siguientes cuatro funciones regresan apuntadores a objetos estáticos que pueden ser sobrescritos por otras llamadas.

`char *asctime(const struct tm *tp)`
asctime convierte la hora de la estructura *tp a una cadena de la forma

```
Sun Jan  3 15:14:13 1988\n\0
```

`char *ctime(const time_t *tp)`
ctime convierte la hora del calendario *tp a hora local; es equivalente a

```
asctime(localtime(tp))
```

`struct tm *gmtime(const time_t *tp)`
gmtime convierte la hora del calendario *tp a Hora Coordinada Universal (*Coordinated Universal Time* —UTC). Regresa NULL si UTC no está disponible. El nombre gmtime tiene significado histórico.

`struct tm *localtime(const time_t *tp)`
localtime convierte la hora del calendario *tp a hora local.

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *tp)
```
strftime da formato a la hora y fecha de *tp en s de acuerdo con fmt, que es análogo al formato printf. Los caracteres ordinarios (incluyendo la terminación '\0') se copian dentro de s. Cada %c se reemplaza como se describe abajo, utilizando los valores apropiados del entorno. En s no se colocan más de smax caracteres. strftime regresa el número de caracteres, excluyendo el '\0', o cero si fueron producidos más de smax caracteres.

%a	nombre abreviado del día de la semana.
%A	nombre completo de la semana.
%b	nombre abreviado del mes.
%B	nombre completo del mes.
%c	representación local de fecha y hora.
%d	día del mes (01-31).
%H	hora (reloj de 24 horas) (00-23).
%I	hora (reloj de 12 horas) (01-12).
%j	día del año (001-366).
%m	mes (01-12).
%M	minuto (00-59).
%p	equivalencia local de AM o PM.
%S	segundos (00-59).
%U	número de semana del año (domingo es el primer día de la semana) (00-53).
%w	día de la semana (0-6, domingo es 0).
%W	número de semana del año (lunes es el primer día de la semana) (00-53).
%x	representación local de la fecha.
%X	representación local de la hora.
%y	año sin el siglo (00-99).
%Y	año con el siglo.
%Z	nombre de la zona horaria, si existe.
%%	%.

B11. Límites definidos en la implantación: <limits.h> y <float.h>

El header <limits.h> define constantes para el tamaño de los tipos enteros. Los valores mostrados son magnitudes mínimas aceptables; se pueden emplear valores mayores.

CHAR_BIT	8	bits en un char
CHAR_MAX	UCHAR_MAX *or* SCHAR_MAX	valor máximo de char
CHAR_MIN	0 *or* SCHAR_MIN	valor mínimo de char
INT_MAX	+32767	valor máximo de int
INT_MIN	-32767	valor mínimo de int
LONG_MAX	+2147483647L	valor máximo de long
LONG_MIN	-2147483647L	valor mínimo de long
SCHAR_MAX	+127	valor máximo de signed char
SCHAR_MIN	-127	valor mínimo de signed char
SHRT_MAX	+32767	valor máximo de short
SHRT_MIN	-32767	valor mínimo de short
UCHAR_MAX	255U	valor máximo de unsigned char
UINT_MAX	65535U	valor máximo de unsigned int

`ULONG_MAX`	`4294967295UL`	valor máximo de **unsigned long**
`USHRT_MAX`	`65535U`	valor máximo de **unsigned short**

Los nombres de la tabla siguiente, subconjunto de **<float.h>**, son constantes relacionadas con la aritmética de punto flotante. Cuando se da un valor, representa la magnitud mínima para la cantidad correspondiente. Cada implantación define los valores apropiados.

`FLT_RADIX`	2	radical de la representación exponencial, p. ej., 2, 16
`FLT_ROUNDS`		modo de redondeo de punto flotante para adición
`FLT_DIG`	6	dígitos decimales de precisión
`FLT_EPSILON`	1E-5	menor número x tal que $1.0 + x \neq 1.0$
`FLT_MANT_DIG`		número de dígitos de base `FLT_RADIX` en la mantisa
`FLT_MAX`	1E+37	máximo número de punto flotante
`FLT_MAX_EXP`		máximo n tal que `FLT_RADIX`$^n-1$ es representable
`FLT_MIN`	1E-37	mínimo número normalizado de punto flotante
`FLT_MIN_EXP`		mínimo n tal que 10^n es un número normalizado
`DBL_DIG`	10	dígitos decimales de precisión
`DBL_EPSILON`	1E-9	menor número x tal que $1.0 + x \neq 1.0$
`DBL_MANT_DIG`		número de dígitos de base `FLT_RADIX` en la mantisa
`DBL_MAX`	1E+37	máximo número **double** de punto flotante
`DBL_MAX_EXP`		máximo n tal que `FLT_RADIX`$^n-1$ es representable
`DBL_MIN`	1E-37	mínimo número **double** normalizado de punto flotante
`DBL_MIN_EXP`		mínimo n tal que 10^n es un número normalizado

APENDICE C: Resumen de modificaciones

Desde la publicación de la primera edición de este libro, la definición del lenguaje C ha sufrido modificaciones. Casi todas fueron extensiones al lenguaje original, y fueron diseñadas cuidadosamente para permanecer compatibles con la práctica existente; algunas repararon ambigüedades de la descripción original, y otras representan modificaciones de la práctica existente. Muchas de las nuevas características se anunciaron en los documentos que acompañan a los compiladores disponibles de AT&T, y posteriormente se han adoptado por otros proveedores de compiladores del lenguaje C. Recientemente, el comité ANSI incorporó más de esos cambios estandarizando el lenguaje, y también introdujo otras modificaciones significativas. Su reporte fue en parte anticipado por algunos compiladores comerciales aún antes de la publicación del estándar formal.

Este apéndice resume las diferencias entre el lenguaje definido por la primera edición de este libro, y lo esperado como la definición del estándar final. Trata solamente al lenguaje en sí, no a su entorno ni a su biblioteca; aunque esas son partes importantes del estándar, hay poco con qué compararlas, puesto que en la primera edición no se intentó definirlas.

- El preprocesamiento está definido más cuidadosamente en el Estándar que en la primera edición, y está extendido: está explícitamete basado en *tokens* (símbolos); existen nuevos operadores para la concatenación de *tokens* (**##**) y creación de cadenas (**#**); hay nuevas líneas de control como **#elif** y **#pragma**; está explícitamente permitida la redeclaración de macros por la misma secuencia de *tokens*; ya no se reemplazan los parámetros que están dentro de cadenas. La separación de líneas por \ está permitida en cualquier lugar, no sólo en definiciones de cadenas y macros. Véase §A12.
- El significado mínimo el más pequeño de todos los identificadores internos se incrementó a 31 caracteres; permitido para identificadores con liga externo permanece en 6 letras, sin importar sin son mayúsculas o minúsculas (muchas implantaciones proporcionan más).
- Las secuencias trigráficas introducidas por ?? permiten la representación de caracteres que no se encuentran en algunos conjuntos. Están definidos los escapes para #\^[ ]{ }|~. Véase §A12.1. Obsérvese que la introducción de trigrafos puede cambiar el significado de cadenas que contengan la secuencia ??.
- Se introdujeron nuevas palabras reservadas (**void**, **const**, **volatile**, **signed**, **enum**). La palabra reservada **entry**, que nunca se puso en uso, fue retirada.
- Se definen nuevas secuencias de escape para uso dentro de constantes de carácter y cadenas literales. El efecto de seguir \ con un carácter que no sea parte de una secuencia de escape aprobada está indefinido. Véase §A2.5.2.

- El trivial cambio favorito de todos: 8 y 9 no son dígitos octales.
- El estándar introduce un conjunto más grande de sufijos para hacer explícito el tipo de constantes: **U** o **L** para enteros, **F** o **L** para flotantes. También afina las reglas para el uso de constantes sin sufijo (§A2.5).
- Las cadenas literales adyacentes se concatenan.
- Existe una notación para cadenas literales amplias de caracteres y constantes de carácter. Véase §A2.6.
- Los caracteres, así como otros tipos, pueden ser explícitamente declarados para tener o no signo, utilizando las palabras reservadas **signed** o **unsigned**. Se retiró la locución **long float** como un sinónimo para **double**, pero **long double** puede ser utilizada para declarar una cantidad flotante de precisión extra.
- Por algún tiempo, el tipo **unsigned char** ha estado disponible. El estándar introduce la palabra reservada **signed** para hacer explícito el signo para **char** y otros objetos enteros.
- Por algunos años, el tipo **void** ha estado disponible en algunas implantaciones. El estándar introduce el uso del tipo **void *** como un tipo de apuntador genérico; anteriormente **char *** desempeñó este papel. Al mismo tiempo, se crearon reglas explícitas contra la mezcla de apuntadores y enteros, y de apuntadores de diferente tipo, sin el uso de operadores **cast**.
- El estándar fija mínimos explícitos en los rangos de tipos aritméticos y delega a los archivos *header* (<**limits.h**> y <**float.h**>) el dar las características de cada implantación particular.
- Las enumeraciones son algo nuevo desde la primera edición de este libro.
- El estándar adopta de C++ la noción de calificador de tipo, por ejemplo, **const** (§A8.2).
- Las cadenas ya no son modificables, por lo que pueden situarse en memoria de sólo lectura.
- Se cambiaron las "convenciones aritméticas usuales", esencialmente de "para enteros, **unsigned** siempre gana; para punto flotante, siempre use **double**" a "promueva al tipo más pequeño de suficiente capacidad". Véase §A6.5.
- Los antiguos operadores de asignación como **=+** realmente desaparecieron. También, los operadores de asignación son ahora *tokens* sencillos; en la primera edición fueron parejas y se podían separar por espacio en blanco.
- Se canceló la licencia del compilador para tratar a los operadores matemáticamente asociativos como computacionalmente asociativos.
- Se introdujo un operador unario **+** por simetría con el **−** unario.
- Un apuntador a una función se puede utilizar como un designador de función sin un operador ***** explícito. Véase §A7.3.2.
- Las estructuras se pueden asignar, pasar a funciones, y regresar por funciones.
- Está permitido aplicar el operador "dirección de" a arreglos, y el resultado es un apuntador al arreglo.
- El operador **sizeof**, en la primera edición, daba el tipo **int**; posteriormente, muchas implantaciones lo hicieron **unsigned**. El estándar hace el tipo explícitamente dependiente de la implantación pero requiere que el tipo **size_t** sea definido en un *header* estándar (<**stddef.h**>). Un cambio semejante ocurre en el tipo (**ptrdiff_t**) de la diferencia entre apuntadores. Véase §A7.4.8 y §A7.7.
- El operador **&** ("dirección de") no se puede aplicar a un objeto declarado **register**, aun si la implantación decide no mantener al objeto en un registro.

- El tipo de una expresión de corrimiento es el del operando de la izquierda; el operando de la derecha no puede promover el resultado. Véase §A7.8.
- El estándar legaliza la creación de un apuntador justo más allá del fin de un arreglo y permite aritmética y relaciones sobre él; Véase §A7.7.
- El estándar introduce (tomándolo de C++) la noción de declaración de una función prototipo que incorpora los tipos de los parámetros, e incluye un reconocimiento explícito de funciones con listas variables de argumentos, junto con una forma aprobada de tratarlas. Véase §§A7.3.2, A8.6.3 y B7. El estilo antiguo todavía se acepta, con restricciones.
- Las declaraciones vacías, que no tienen declaradores y no declaran al menos a una estructura, unión o enumeración, están prohibidas por el estándar. Por otro lado, una declaración con sólo un rótulo de estructura o de unión, redeclara ese rótulo aun si fue declarado en un alcance más externo.
- Están prohibidas las declaraciones externas sin ningún especificador o calificador (con sólo el declarador).
- Algunas implantaciones, cuando examinan una declaración **extern** en un bloque más interno, podían exportar la declaración al resto del archivo. El estándar hace claro que el alcance de tal declaración es sólo el bloque.
- El alcance de los parámetros se introduce en la proposición compuesta de una función, así que las declaraciones de variables en el nivel superior de la función no pueden ocultar los parámetros.
- Los espacios de nombre de los identificadores son algo diferentes. El estándar pone todos los rótulos en un espacio sencillo de nombre, y también introduce un espacio separado de nombres para etiquetas; véase §A11.1. Los nombres de miembros también están asociados con la estructura o unión de la que son parte (esto ha sido práctica común por algún tiempo).
- Las uniones se pueden inicializar; el inicializador se refiere al primer miembro.
- Las estructuras, uniones y arreglos automáticos se puede inicializar, aunque en una forma restringida.
- Los arreglos de caracteres con tamaño explícito se pueden inicializar con una cadena literal con exactamente la misma cantidad de caracteres (el \0 se excluye calladamente).
- La expresión de control y las etiquetas de las alternativas de un **swith**, pueden tener cualquier tipo entero.

Indice

<assert.h> 278
' carácter apóstrofo 21, 41-43, 213
\f carácter avance de hoja (formfeed) 42, 213
" carácter comillas 8, 21, 43, 214
\a carácter de alarma 42, 213
\b carácter de retroceso 8, 42, 213
\t carácter de tabulación 8, 11, 42, 213
\v carácter de tabulación vertical 42, 213
\\carácter diagonal invertida (antidiagonal) 8, 42
\n carácter nueva línea 7, 16, 21, 41-42, 213, 265
\o carácter nulo 32, 42, 213
\r carácter retorno de carro 42, 213
_carácter subguión 39, 212, 265
0x... constante hexadecimal 41, 213
0... constante octal 41, 213
#define 15, 98-99, 254
#define con argumentos 99
#define de varias líneas 98
#define vs. enum 43-44, 165
#else, #elif 101, 256
#endif 101
#error 258
?: expresión condicional 57, 230
#if 101, 151, 256
#ifdef 101, 256
#ifndef 101, 256
#include 35, 98, 168, 256
_IOFBF, _IOLBF, _IONBF 267
#line 257
{} llaves 7, 10, 61, 93
& operador AND para bits 53, 228-229
&& operador AND lógico 22, 46, 54, 228-229
-> operador apuntador a estructura 146, 222
, operador coma 69, 231
?: operador condicional 57, 230
+ operador de adición 45-46, 226-227
= operador de asignación 18, 46, 230
+= operador de asignación 55
~ operador de complemento a uno 54, 224-225
>> operador de corrimiento a la derecha 54, 227-228
<< operador de corrimiento a la izquierda 54, 227-228
-- operador de decremento 19, 51, 118, 224
!= operador de desigualdad 17, 45, 228
& operador de dirección 104, 225
/ operador de división 11, 45-46, 226
== operador de igualdad 20-21, 45-46, 228
++ operador de incremento 19, 51, 118, 224
* operador de indirección 104, 225
* operador de multiplicación 45-46, 226-227
! operador de negación lógica 224-225
- operador de substracción 45-46, 226
operador del preprocesador 100, 254
operador del preprocesador 100, 254
+ operador más unario 224-225
>= operador mayor o igual que 46, 228
> operador mayor que 46, 227-228
<= operador menor o igual que 46, 228
< operador menor que 46, 227-228
- operador menos unario 224-225
. operador miembro de estructura 142, 222
% operador módulo 45, 226
^ operador OR exclusivo para bits 53, 229
| operador OR inclusivo para bits 53, 229
|| operador OR lógico 22, 46, 54, 229
#pragma 258
\x secuencia de escape hexadecimal 42, 213
#undef 99-100, 190, 254
++ y -- postfijos 51, 116
++y -- prefijos 51, 118

a.out 6, 79
acceso a un archivo 176, 187-188, 196, 266
agregar a un archivo 177, 193, 266
alcance (scope) 215, 251-252
alcance de automáticas 89, 252
alcance de externas 89, 252
... declaración 172, 223
alcance de una etiqueta 72, 246, 252-253
alcance léxico 251
alineación de campos de bits 165-166, 236
alineación por unión 205
ambigüedad en if-else 62, 247, 259
American National Standards Institute (ANSI) ix, 2, 211
apertura de un archivo 176-177, 187-188, 190
apuntador a función 130-131, 162, 222
apuntador a una estructura 151
apuntador a una función 130-131, 162, 222
apuntador genérico véase apuntador void*

apuntador nulo 113, 219
apuntador para archivo 177, 194, 226
apuntador void* 103, 114, 132-133, 219-220
apuntador vs. arreglo 108-110, 115-116, 124-125
apuntadores y subíndices 108, 109, 240
árbol binario 154
árbol de descripción gramatical 135
archivo encabezado 35, 90-91
archivo para inclusión dir.h 202
archivo para inclusión fcntl.h 191
archivo para inclusión stat.h 199-200
archivo para inclusión syscalls.h 189
archivo para inclusión types.h 200, 202
argumento apuntador 110-111
argumento de función 27, 223
argumento efectivo *véase* definición de argumento
argumentos de un programa *véase* línea de comandos como argumentos
argumentos tomados de una línea de comandos 126-130
aritmética de apuntadores 104-105, 108-109, 111-114, 129, 152-153, 226-227
aritmética de direcciones *véase* aritmética de apuntador
aritmética ilegal de apuntadores 113-114, 152-153, 227
aritmética usual de conversiones 47, 218
arreglo bidimensional 122, 123, 243
arreglo de apuntadores 118-119
arreglo de caracteres 21, 30, 115-116
arreglo de estructuras 147, 162-163
arreglo multidimensional 122, 239-240
arreglo vs. apuntador 108-111
asignación múltiple 22
asignador de almacenamiento 157, 204-208
asociatividad de los operadores 57, 220

bloque *véase* proposición compuesta
BUFSIZ 267

cadena de caracteres *véase* constante cadena
cadena literal *véase* constante cadena
cadena nula 43
cadena vacía 42
cálculo de años bisiestos 45, 122
campo véase *Carácter de alarma*
carácter de control 273
carácter de impresión 273
carácter signado 15, 216
carácter sin signo 49, 216
caracteres espacio en blanco 174, 183-184, 269, 273
categoría de almacenamiento 215
categoría de almacenamiento automática 34, 215
categoría de almacenamiento estática 34, 92, 215
categoría de almacenamiento automática 33, 215
ciclo *véase* While, for
ciclo infinito for(;;) 66-67, 99
CLK_TCK 280
comando cc 6, 79
comando ls 198
comentario 9, 211-212, 254
comparación de apuntadores 113, 152-153, 207, 228-229
compilación condicional 100-101, 256
compilación de varios archivos 79
compilación separada 75, 88-89, 251
compilando un programa en C 6, 27
concatenación de cadenas 43, 100, 214-215
concatenación de símbolos 100, 254
condición de límite 20, 71-72
conducto (pipe) 168, 188
conjunto de caracteres 253
conjunto de caracteres ASCII 21, 42, 48, 253, 273
conjunto de caracteres EBCDIC 48
conjunto de caracteres ISO 253
constante cadena 7, 21, 32, 42-43, 110, 115, 214-215
constante cadena amplia (wide) 214
constante carácter 21, 41, 213-214
constante carácter amplio (wide) 213-215
constante carácter en octal 41-42
constante de enumeración 43-44, 101, 213-214, 238
constante double 41, 214
constante entera 13, 41, 213
constante flotante 13, 41, 214
constante long 41, 213
constante long double 41, 214
constante unsigned 41, 213
constante unsigned long 41, 213
constantes 41, 212-213
contenido de <stdio.h> 194
continuación de líneas 253
convenciones léxica 211
conversión 217-220
conversión %ld 19-20
conversión apuntador-entero 219-220, 226
conversión carácter-entero 24-25, 47, 218
conversión de apuntadores 157, 219, 227
conversión de argumentos de función *véase* promoción de un argumento
conversión de fecha 122
conversión de nombre de un arreglo 109-110, 221
conversión de tipo por return 81, 249
conversión de una función 221
conversión de nombre de arreglo 109-110, 221
conversión double-float 50-51, 218
conversión entero-apuntador 219, 227
conversión entero-carácter 50
conversión entero-flotante 13, 218
conversión float-double 49, 218
conversión flotante-entero 50, 218
conversión por cast 50-51, 219-220, 226
conversión por asignación 49, 230
conversión por return 81, 249
conversiones aritméticas usuales 47, 218
creación de un archivo 177-178, 187-188
CRLF 167, 265
cualificador const 45, 217, 234
cualificador de tipo 229-230, 234

cualificador volatile 217, 234
cuenta de argumentos argc 126

decisión con varias alternativas 226-227
declaración 9, 44-45, 232-242
...declaración 172, 223
declaración de apuntadores 104, 110-111, 239
declaración de campos de bits 165-166, 235
declaración de categorías de almacenamiento 232-233
declaración de función 240-241
declaración de función static 92
declaración de tipo 238-239
declaración de tipos inconsistentes 80
declaración de un arreglo 24, 122, 239
declaración de una estructura 142, 234-235
declaración de unión 162-163, 234-235
declaración de variable externa 33, 249
declaración externa 249-251
declaración implícita de una función 29, 81, 222
declaración inconsistente de tipo 80
declaración static de una función 92
declaración typedef 161, 232, 244
declaración vs. definición 35-36, 89
declarador 238-242
declarador abstracto 244
declarador de arreglo 239
declarador de función 240
definición de almacenamiento 232-233
definición de argumentos 27, 222
definición de un argumento 27, 222
definición de un parámetro 27, 222
definición de una función 27, 77, 249
definición de una macro 254
definición de una variable externa 35-36, 251
definición tentativa 251
desbordamiento 46, 221, 274, 279
descriptor de archivo 188
descriptor gramatical recursivo-descendente 136
designativo de función 222
dirección de un registro 232
dirección de una variable 30, 104, 224
división entera 11, 45
EDOM 274

efectos colaterales 59, 99, 220, 223
eficiencia 56, 92, 97-98, 157, 207
ele *véase* proposición if-else
else-if 25, 63
encabezado <ctype.h> 48, 272
encabezado <errno.h> 272
encabezado <float.h> 41, 281-282
encabezado <limits.h> 41, 281-282
encabezado <locale.h> 265
encabezado <math.h> 49, 79, 274
encabezado <setjmp.h> 278
encabezado <signal.h> 279
encabezado <stdarg.h> 172, 192, 278
encabezado <stddef.h> 114, 150, 265
encabezado <stdio.h> 6, 17, 98-99, 113, 167-168, 265
encabezado <stdlib.h> 157, 275
encabezado <string.h> 43, 117, 273
encabezado <time.h> 279
entrada del teclado 16, 167, 188-189
entrada estándar 167, 177-178, 188-189
entrada sin uso de buffer 189
entrada usando buffer 189
entrada y salida por terminal 16
entrada/salida de caracteres 16, 167
enum vs. #define 44, 165
enumerador 214, 237-238
EOF 17, 168, 266
equivalencia de tipos 245
ERANGE 274
errno 272, 274
error estándar 177-178, 188-189
errores de entrada/salida 181, 272
ecalamiento en aritmética de apuntadores 113-114, 218-219
espacio de nombres 252
espacio en blanco 211
especificador de categoría de almacenamiento 232-233
especificador de categoría de almacenamiento auto 232-233
especificador de categoría de almacenamiento extern 33, 35-36, 88-89, 232, 233
especificador de categoría de almacenamiento register 92, 233
especificador de categoría de almacenamiento static 92, 233
especificador de categoría extern 33, 35-36, 88-89, 232-233
especificador de tipo 233-234
especificador emum 43-44, 237
especificador struct 234-235
especificador unión 234-235
estructura anidada 143-144
estructura autorreferida 155, 235
estructura de un bloque 61, 93, 246-247
estructura DIR 199
estructura Dirent 199
estructura stat 199
estructuras anidadas 143-144
estructuras mutuamente recursiva 155, 235
etapas de traducción 211, 252
etiqueta 72, 245-246
etiqueta por omisión 64, 246
etiquetas del case 64, 246
evitando el goto 72-73
excepciones (condiciones excepcionales) 220-221, 279
EXIT_FAILURE, EXIT_SUCCESS 277
expansión de macros 254-255
exponenciación 26, 275
expresión 220-232
expresión constante 42, 64, 101, 230-231
expresión de asignación 18, 22, 56, 230
expresión entre paréntesis 221

expresión primaria 221
expresión proposición 61, 63, 245-246
extensión del signo 49-50, 195, 214

falta de especificador de categoría de almacenamiento 233
falta de especificador de tipo 233-234
FILENAME_MAX 266
fin de archivo *véase* EOF
FOPEN_MAX 266
for vs. while 14-15, 66-67
forma de un inicializador 94, 231
formato de un programa 11, 20-21, 24-25, 44-45, 152-153, 212
función addpoint 144
función addtree 156
función afree 113
función al nuevo estilo 223
función al viejo estilo 28, 36, 80-81, 223
función alloc 111-112
función atof 79-80
función atoi 48, 68, 81
función bisearch 64, 148-149, 151-152
función bitcount 56
función canonrect 145
función closedir 203
función copy 31, 35
función day_of_year 122-123
función del 135-136
función de biblioteca 7, 75, 88
función de biblioteca abort 277
función de biblioteca abs 277
función de biblioteca acos 275
función de biblioteca asctime 280
función de biblioteca asin 275
función de biblioteca atof 275
función de biblioteca atoi 48, 68, 81
función de biblioteca atol 276
función de biblioteca bsearch 277
función de biblioteca calloc 184, 276
función de biblioteca ceil 275
función de biblioteca clearerr 272
función de biblioteca clock 280
función de biblioteca cos 275
función de biblioteca cosh 275
función de biblioteca ctime 280
función de biblioteca difftime 280
función de biblioteca div 277
función de biblioteca exit 179-180, 277
función de biblioteca exp 275
función de biblioteca fabs 275
función de biblioteca fclose 179, 266
función de biblioteca feof 181, 272
función de biblioteca ferror 195
función de biblioteca fflush 266
función de biblioteca fgetc 270
función de biblioteca fgetpos 272
función de biblioteca floor 275
función de biblioteca fmod 275
función de biblioteca fopen 177, 266
función de biblioteca fprintf 178, 267
función de biblioteca fputc 271
función de biblioteca fputs 181, 271
función de biblioteca free 184-185, 277
función de biblioteca freopen 179, 266
función de biblioteca frexp 275
función de biblioteca fscanf 178, 269
función de biblioteca fseek 272
función de biblioteca fsetpos 272
función de biblioteca ftell 272
función de biblioteca fwrite 271
función de biblioteca getbits 54
función de biblioteca getc 178, 271
función de biblioteca getchar 16, 167, 178, 271
función de biblioteca getenv 277
función de biblioteca gets 181, 271
función de biblioteca gmtime 280
función de biblioteca isalnum 151, 273
función de biblioteca isalpha 150, 183, 273
función de biblioteca iscntrl 273
función de biblioteca isdigit 183, 273
función de biblioteca isgraph 273
función de biblioteca islower 183, 273
función de biblioteca isprint 273
función de biblioteca ispunct 273
función de biblioteca isspace 150, 183, 273
función de biblioteca isupper 183, 273
función de biblioteca isxdigit 273
función de biblioteca labs 277
función de biblioteca ldexp 275
función de biblioteca ldiv 277
función de biblioteca localtime 280
función de biblioteca longjmp 279
función de biblioteca malloc 158, 184, 276
función de biblioteca memchr 274
función de biblioteca memcmp 274
función de biblioteca memcpy 274
función de biblioteca memmove 274
función de biblioteca memset 274
función de biblioteca mktime 280
función de biblioteca modf 275
función de biblioteca perror 272
función de biblioteca pow 26, 275
función de biblioteca printf 7-8, 11-12, 19, 169, 269
función de biblioteca putc 178, 271
función de biblioteca putchar 16, 168, 178, 271
función de biblioteca puts 181, 271
función de biblioteca qsort 277
función de biblioteca raise 279
función de biblioteca rand 276
función de biblioteca realloc 276
función de biblioteca remove 267
función de biblioteca rename 267
función de biblioteca rewind 272
función de biblioteca scanf 107, 173, 270
función de biblioteca setbuf 267
función de biblioteca setjmp 278

función de biblioteca setvbuf 267
función de biblioteca signal 279
función de biblioteca sin 275
función de biblioteca sinh 275
función de biblioteca sprintf 171, 269
función de biblioteca sqrt 275
función de biblioteca srand 276
función de biblioteca sscanf 270
función de biblioteca strcat 273
función de biblioteca strchr 273
función de biblioteca strcmp 273
función de biblioteca strcpy 273
función de biblioteca strcspn 274
función de biblioteca strerror 274
función de biblioteca strftime 280
función de biblioteca strlen 274
función de biblioteca strncat 273
función de biblioteca strncmp 273
función de biblioteca strncpy 273
función de biblioteca strpbrk 274
función de biblioteca strrchar 273
función de biblioteca strspn 274
función de biblioteca strstr 274
función de biblioteca strtod 275
función de biblioteca strtok 274
función de biblioteca system 184, 277
función de biblioteca tan 275
función de biblioteca tanh 275
función de biblioteca time 280
función de biblioteca tmpfile 267
función de biblioteca tmpnam 267
función de biblioteca tolower 169, 183, 273
función de biblioteca toupper 183, 273
función de biblioteca ungetc 183-184, 271
función dirdel 136-137
función dirwalk 200-201
función error 192
función fgets 181
función filecopy 178-179
función fopen 195-196
función fputs 181
función free 207-208
función fsize 201
función getch 87
función getint 107
función getline 31, 34, 77, 182
función getop 86
función gettoken 137
función getword 150-151
función hash 159-160
función install 160
función itoa 70
función lookup 160
función lower 48
función main 6
función makepoint 144
función malloc 206-207
función month_day 122-123
función month_name 124
función morecore 207
función numcmp 133
función opendir 202
función pop 86
función power 26-27, 28-29
función printd 96
función ptinrect 145
función push 85-86
función qsort 96-97, 121, 132
función que no hace nada 78
función rand 51
función readdir 203
función readlines 120
función reverse 69
función shellsort 68
función squeeze 52
función srand 51
función strcat 53
función strcmp 117
función strcpy 116-117
función strindex 77
función strlen 43, 110, 114
función swap 106, 121, 133
función talloc 157
función treeprint 156-157
función trim 71
función ungetc 87
función vacía 78
función writelines 120-121
función _fillbuf 196-197
funciones de biblioteca atan, atan2 275
funciones de biblioteca atexit 277
funciones de biblioteca log, log10 275
funciones de biblioteca strtol, strtoul 276
funciones de biblioteca vprintf, vfprintf, vsprintf 192, 269
funciones de prueba sobre caracteres 183, 272

generación de un apuntador 221
getchar sin uso de buffer 189
getchar usando buffer 190

Hoare, C. A. R. 96
HUGE_VAL 274

identificador 212
inclusión de archivos 98, 256
índices negativos 111
inicialización 44-45, 94, 242
inicialización de apuntadores 113, 152
inicialización de arreglos bidimensionales 123, 243
inicialización de arreglos de estructuras 147-148
inicialización de automáticas 33, 45, 94, 242
inicialización de estáticas 45, 94, 242
inicialización de externas 45, 90, 94, 242
inicialización de un arreglo 95, 124, 243
inicialización de un arreglo bidimensional 123, 243
inicialización de una constante cadena 95, 242
inicialización de una estructura 142, 242
inicialización de una unión 243

inicialización dentro de un bloque 93, 246-247
inicialización por cadena constante 95, 242
inicialización por omisión 95, 242
inicializador 251

legibilidad de un programa 11, 56, 70-71, 95, 162
ligado 215, 251-252
ligado externo 82, 212, 216, 233, 252
ligado interno 216, 252
#line 216, 252
línea de comandos como argumentos 126, 131
línea de control 98, 254-258
lista de argumentos de longitud variable 171, 192, 223, 241, 278, 279
lista de argumentos **void** 35, 81, 241, 249
lista de palabras clave 212
longitud de constantes simbólicas 39
longitud de nombres 39-40, 212
longitud de nombres de función 40, 212
longitud de nombres de variable 212
longitud de nombres de variables 212
longitud de nombres externos 40, 212
longitud de nombres internos 40, 212
longitud de una cadena 32, 43, 115
LONG_MAX, LONG_MIN 276

llamada al sistema **close** 192
llamada al sistema **creat** 190
llamada al sistema **fstat** 202
llamada al sistema **lseek** 193
llamada al sistema **open** 190
llamada al sistema **read** 188
llamada al sistema **sbrk** 207
llamada al sistema **stat** 199
llamada al sistema **unlink** 193
llamada al sistema **write** 188
llamada por referencia 29
llamada por valor 29, 106, 223
llamadas al sistema 187

macro **feof** 195
macro **ferror** 195
macro **getc** 195
macro **putc** 195
macros con argumentos 98-99
manifestación de constantes 254
modismos en manipulación de bits 54, 164-165
modo de acceso a un archivo 176-177, 196-197, 266
modularización 26, 30, 36, 75, 82-84, 119

nodo-i (**inode**) 198
nombre 212
nombre de tipo **clock_t** 279
nombre de tipo **FILE** 176
nombre de tipo **fpos_t** 272
nombre de tipo **ptrdiff_t** 114, 162, 227
nombre de tipo **size_t** 114, 150, 162, 226, 266
nombre de tipo **time_t** 279
nombre de tipo **wchar_t** 214
nombre de un arreglo como argumento 30, 111, 123
nombre de un miembro de estructura 142-143, 235-236
nombre __FILE__ para el preprocesador 278
nombre __FILE__ para el preprocesador 278
nombre __LINE__ para el preprocesador 278
nombres de tipo div_t y ldiv_t 277
nombres de tipos 244
nombres predefinidos para el preprocesador 258
notación científica 41, 81-82
notación E 41, 214
notación Polaca 82
notación Polaca inversa 82
notación sintáctica 215
nueva línea 211, 253-254
NULL 113
números mágicos 15

objeto 215, 218
ocultamiento de información 75-76, 84, 86
ocultamiento de un nombre 103
omisión de la prueba contra cero 62, 117
operaciones permitidas sobre apuntadores 114
operaciones sobre uniones 163
operador **cast** 50, 157, 184, 218, 226, 244
operador de dirección 103, 225
operador **defined** para el preprocesador 101, 257
operador OR lógico 22, 45, 54, 229
operador **sizeof** 101, 114, 150, 225-226, 271
operadores aditivos 226
operadores aritméticos 45
operadores de adición 227
operadores de asignación 46, 55, 229
operadores de bits 53, 228
operadores de corrimiento 53, 227
operadores de igualdad 45, 228
operadores de relación 17, 45, 227
operadores multiplicativos 227
operadores para bits 53, 229
orden de almacenamiento de un arreglo 124, 240
orden de evaluación 22, 54, 59, 69, 86, 100, 105, 221
orden de evaluación de una expresión 58, 221
orden de traducción 253
ordenamiento lexicográfico 130
ordenamiento numérico 130
ordenando líneas de texto 118, 131
O_RDINDLY, O_RDWR, O_WRONLY 190

palabra clave **asm** 212
palabra clave **fortran** 212
palabras reservadas 38, 212
parámetro 93, 110, 222
parámetro formal *véase* parámetro
permisos sobre un archivo 191
portabilidad 3, 39, 47, 54, 162, 167, 169, 204
posición de llaves 11
precedencia de los operadores 18, 58, 105, 147-148, 220

preprocesador de macros 97, 252-258
problemas regionales 265
programa **cat** 176, 178-179
programa **dcl** 146
programa de búsqueda en una tabla 158
programa de conversión a minúsculas 169
programa de conversión de temperaturas 8-10, 13-15, 16
programa de la calculadora 81, 83, 85, 174
programa de ordenamiento 119, 242
programa de terminación 178, 180
programa **echo** 127, 128
programa **fsize** 200
programa de concatena archivos 176
programa que copia archivos 17, 18, 189, 191
programa que cuenta caracteres 19
programa que cuenta espacios en blanco 23, 65
programa que cuenta líneas 20
programa que cuenta palabras 21, 154
programa que cuenta palabras clave 148
programa que encuentra la línea más larga 31, 34
programa que encuentra un patrón 75, 77, 128-129
programa que escribe la lista de un directorio 198
programa **undcl** 149
programa a la defensiva 63, 65
promoción de un argumento 49, 224
promoción integral 48, 218
proposición **break** 65, 69, 249
proposición compuesta 61, 93, 247, 250-251
proposición **continue** 70, 249
proposición de asignación anidada 18, 22, 57
proposición de selección 248
proposición **do** 68, 249
proposición **for** 14, 19, 66, 249
proposición **goto** 70, 249
proposición **if else** 20, 22, 61, 248
proposición nula 19, 247
proposición **return** 27, 33, 76, 79, 250
proposición **switch** 64, 82, 248
proposición vacía *véase* proposición nula
proposición **while** 10, 66, 249
proposiciones 247-250
proposiciones de iteración 249
proposiciones de salto 249
proposiciones etiquetadas 70, 247
prototipo de una función 28, 33, 49, 78, 132, 222
punto y coma 10, 16, 19, 61, 63
quicksort 96, 121

RAND_MAX 276
recursividad 95, 154, 156, 201, 224, 293
redirección *véase* redirección de entrada/salida
redirección de entrada/salida 168, 177, 186
redirección de salida 168
referencia de un arreglo 223
reglas de alcance 89, 251
reglas de conversión de tipo 46, 48, 218
regreso de lo que ha entrado 84
remoción de una definición *véase* **#undef**
remplazo de símbolos 253
reservación de almacenamiento 231
restricciones de alineación 153, 157, 164, 179, 204, 219
return de **main** 27, 176
Richards, M. 1
Ritchie, D. M. xi
rótulo de enumeración 236
rótulo de estructura 152, 233
rótulo de unión 233

salida en pantalla 16, 168, 179, 186
salida estándar 168, 177, 186
sangrado 10, 20, 24, 62
secuencia de escape 8, 20, 39-41, 213, 254
secuencia de proposiciones 247
secuencia trigráfica 254
SEEK_CUR, SEEK_END, SEEK_SET 272
semántica de la llamada a una función 223
semántica de la referencia a una estructura 224
Shell, D. L. 67
SIG_DFL, SIG_ERR, SIG_IGN 279
símbolo (**token**) 212, 254
sin desbordamiento 46, 274, 279
sintaxis de la llamada a una función 221
sintaxis de la referencia a una estructura 222
sintaxis de nombres de variable 37, 212
sistema de archivos de **UNIX** 187, 198
stderr 177, 179, 266
stdin 177, 266
stdout 177, 266
stream binario 176, 265-266
stream de texto 16, 167, 265
subarreglos como argumentos 111
subíndices negativos 111
subíndices y apuntadores 107, 110, 241
subindizado de arreglos 23, 107, 221, 241
substracción de apuntadores, 114, 153, 218
sufijo.h para nombre de archivo 35
sufijos en constantes 39, 213
supresión de asignación, **scanf** 173, 269

tabla de conversiones de **printf** 170, 268
tabla de conversiones de **scanf** 174, 270
tabla de ejemplos de **printf** 14, 170
tabla de encabezados estándar 266
tabla de **hash** 159
tabla de operadores 59
tabla de secuencias de escape 42, 213
tamaño de números 9, 19, 38, 281
tamaño de una estructura 153, 225
tamaño por omisión de un arreglo 95, 125, 149
terminador de proposición 10, 61
Thompson, K. L. 1
tipo **char** 10, 38, 216, 234
tipo de cadena 221
tipo de constante 39, 214
tipo **double** 10, 19, 38, 217, 234
tipo enumeración 217

tipo float 39, 215
tipo incompleto 235
tipo int 9, 38, 235
tipo long 10, 19, 38, 217, 234
tipo long double 38, 217
tipo por omisión de una función 33, 222
tipo short 10, 38, 234, 235
tipo signed 38, 234
tipo unsigned 38, 56, 217, 234
tipo unsigned char 38, 189
tipo void 33, 216, 219, 234
tipos aritméticos 216
tipos derivados 1, 10, 216
tipos flotantes 216
tipos fundamentales 9, 38, 215
tipos integrales 216
TMP_MAX, 267
truncamiento de punto flotante 49, 217
truncamiento por división 11, 45, 226

ULONG_MAX, 276
unidad de traducíón 211, 249, 251
uso de buffer véase *Setbuf Setvbuf*

valor numérico de expresiones lógicas 48
valor numérico de una expresión de relación 46, 48
valor numérico de una expresión lógica 48
valor−1 217
variable 215
variable automática 34, 82, 216
variable externa 34, 81, 216
variables externas static, 91
variables internas static 91
va_list, va_start, va_arg, va_end 171, 192, 269, 278
vector de argumentos argv 125, 190

while vs. for 15, 66